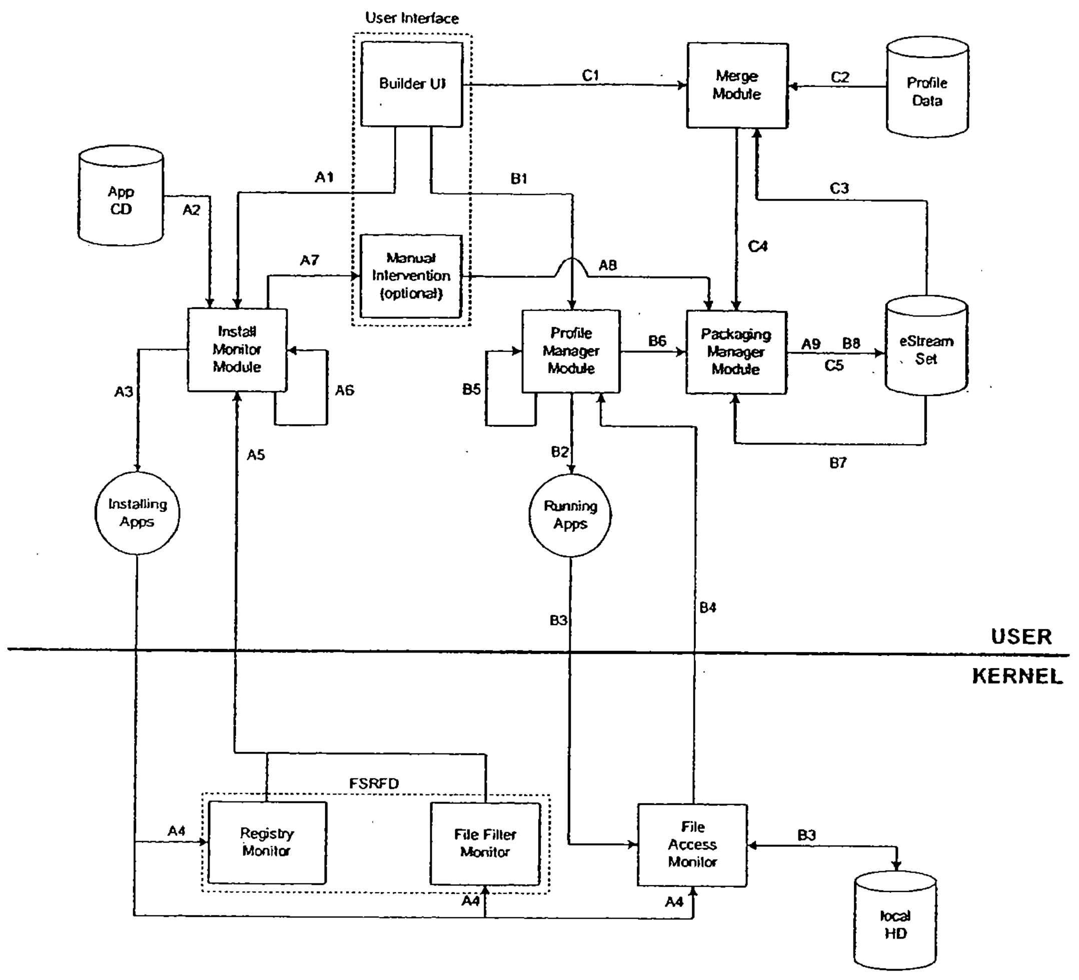
In addition, kernel device drivers are used to actually hook into the operating system to monitor the registry and file changes during installation of the application. This is accomplished by the FSRFD module.

The kernel device driver is also used for gathering monitoring file block references from the operating system to the file system. This is accomplished by the File Access Monitor.

eStream Application Builder High-Level Design Diagram



v 0.1

Interfaces

The interfaces are divided into three use cases: application installation monitoring, application profiling, and merging of the uploaded profile data.

Use Case #1: Install Monitor

A1. Builder UI to Install Monitor – send the name of the application installation executable

A2. App CD to Install Monitor – the CD containing the application is fed into the installation monitor module

A3. Install Monitor to Installation App – invoke the installation program

A4. Installation App to FSRFD – monitor all changes to the registry and files when installation program write to local file system

A5. FSRFD to Install Monitor – send all registry and file changes

A6. Install Monitor to itself – repeat all applications in the suite and merge all data

A7. Install Monitor to Manual Intervention – send a list of registry and file captured by the install monitor to UI and allow user to add or delete any entries

A8. Manual Intervention to Package Manager – send the final registry and file relocation data to the packager

A9. Package Manager to database – data set is packaged into appInstallBlock and the rest of the application files suitable for eStreaming

Use Case #2: Profiling

B1. Builder UI to Profile Manager – send the name of the application executable

B2. Profile Manager to Run App – invoke the application

B3. Run App to eStream File Access Monitor – record sequences of page requests

B4. File Access Monitor to Profile Manager – save the profile information

B5. Profile Manager to Profile Manager – repeat for each application in the suite

B6. Profile Manager to Package Manager – send all profile data for merging into a single data

B7. database to Package Manager – get eStream Set from the database

B8. Package Manager to database – save the updated eStream Set

Use Case #3: Merging Profile data (not supported in version 1.0)

C1. Builder UI to Merger – send the application name with profile data to merge

C2. database to Merger – get uploaded Profile Sequence Matrix from the Profile Server

C3. database to Merger – get the old appInstallBlock from database

C4. Merger to Package Manager – reinsert the Profile data into appInstallBlock

C5. Package Manager to database – save the updated appInstallBlock

Requirements

Please see eStream1.0-REQ.doc for the most up-to-date list of the Builder requirements. This requirement list may not contain the most recent changes. Each requirement is identified by a tag such as R-XXXX for easy references elsewhere in the document.

- **R-Background**: The installation monitor runs in the background, when an eStream application is installed as part of its preparation or building.
- **R-RegistryCapture**: The installation monitor captures all the updates to the System Registry that take place during the install. These updates are captured as a .REG file. Note that registry key deletions are also captured and stored in the .REG file. Please see the LLD doc by Charles Booher about the Registry spoofing database.
- **R-FileCapture:** The installation monitor records all the files created in the two kinds of directories: the install directory (the F_I group described above) and the common directories (the F_{CSU} group). All the files created are copied to the Installation Set and the File Relocation Map (FRM) created for the F_{CSU} group files. <Note: as far as a system common DLL is concerned, the eStream client should (a) overwrite the existing DLL if it exists (b) spoof it if doesn't exist. This is necessary because some installations may depend on newer versions of, say MSVCRT.DLL and in Windows there is no way to maintain different versions of the same DLL>.
- **R-InitialProfiling:** The Builder must be able to gather initial set of application profile data. This data consists of the page access pattern for starting and immediately shutting down an application.
- **R-Packaging:** The Builder must package the eStream Set into a easily manageable packages suitable for ASP administrators to download to their servers. The package can be divided into two sets:

  1. Installation Set - an appInstallBlock which is a set of data needed to setup the client machine for running a particular eStream application. The appInstallBlock is converted into an installation executable for simplifying the initial application set-up on the client machine.
  2. Run-time Set - a set of files associated with a particular application. At run-time, appropriate pages from this set of files is streamed to the client.

- **R-Merging (not supported in version 1.0):** The Builder must be able to collect per-user profile data from the Profile Server and merge the profile data into a combined data usable for updating the profile data in the appInstallBlock. This profile data can also be collected for use by the ASP or application developers.
- **R-NoQuietOperation:** The Builder is not required to be run in an environment where no other applications are running. But, since the Builder operates by invoking application installation program, it inherits any restrictive "Quiet-Operation" requirement from the installation program. Thus, if the installation program of an application has a "Quiet-Operation" requirement, then the "Quiet-Operation" must be enforced by the user when running the Builder.
- **R-AllClient**: The Builder should provide functionality to create installation set(s) for each of the clients eStream 1.0 is going to support. <Preferably there should be only one builder program that should recognize the OS it is running on and should create the appropriate installation set. Also if possible, we should be able to "diff" installation sets for different OSs and if they are same, we should be able to create

a single installation set for those OSs. The clients to be supported are W2K, WinNT4.0 and Win98>.

- **R-AppIdGeneration**: It should be possible to change the appId of the eStream set when an ASP wants to "install" the eStream set in order to host it. Typically the builder will generate a default appId number for a new application which can be overridden by the ASP installer by using a Builder tool.
- **R-SuiteSupport**: It should be possible to create a merged eStream set for a suite of applications. E.g. Office consisting of Word, Excel and Powerpoint. This could be done either by providing a tool for merging multiple eStream sets or by allowing the builder to serially monitor multiple installations in a session and then allowing the user to create a single package at the end of the session.
- **R-Testing**: It should be possible to test the Builder using a stand-alone tester and not require the eStream client+server programs.
- **R-UpgradeSupport**: The appInstallBlock should have support for indicating upgrades at the support site. E.g. When an eStream application is upgraded at the server (not as a separate app), the client will no longer be able to access/use it. We should provide some version of the appInstallBlock itself so that clients should detect that they will need to download the appInstallBlock again.
- **R-ManualIntervention**: In the process of creating an eStream set it should be possible for the user to delete file entries and registry entries manually to "trim" the eStream set if she so desires assuming the user knows what she is doing.

Issues

- Profile Sequence Matrix is different for different machine configuration even if the user's usage pattern is the same.
- Profile Sequence Matrix doesn't contain the right successor profile information as eStream cache is warmed up and pages from the cache is replaced.
- Merging Module must take different machine configuration into account. Should this information be uploaded by the client as the same time it uploads the Profile Sequence Matrix to the Profile Server?
- What is the difference between profiling based on the page sequencing seen by the eStream Cache Manager versus the page sequencing missed by the eStream Cache Manager?

eStream Application Builder Interfaces

Authors: Sanjay Pujare and David Lin

Version 0.1

[redacted]

Sub-components of the Builder covered here (note, these are logical divisions and not necessarily physical):

1) Builder UI (the only program directly invoked by the user).
2) Install Monitor module.
3) File System and Registry filter driver (FSRFD).

Interfaces:

1) User to Builder UI:

   There are multiple Use Cases represented by this interface:

   1. Monitor a new install: user needs to provide the (a) setup.exe path and (b) the destination drive where the app will be installed. If this is the second or subsequent install in the session then destination drive should be same as previous ones. After the Builder has finished running setup.exe the user needs to tell the Builder whether the installation was successful or not (unless the Builder can figure it out from the exit code of the setup.exe process). If the installation was not successful, Builder will erase all the installation set data created.

      Validation: Check that the destination drive is not the same as the system drive (e.g. C: on most systems), since the install monitor won't be able to differentiate between common files and installation files in that case.

      Note: The Builder user needs to start with a machine that is "pristine" i.e. this machine should only have the OS installed and no other application or data files. This way we get "maximal" installation to cover all the client configurations i.e. the application install will not be affected by any existing applications or registry settings etc. <At this point it is not clear if "pristine" OS includes any service packs>.

   2. Manual Edits: This allows the user to manually delete file entries and registry entries. Normally this should not be used. This functionality provides a screen that allows entry deletions for registry/files.
   3. Initial Profiling: This allows the initial profile to be created for the application(s) installed in this session. <User needs to provide the .EXE that will be run for the initial profile??>.
   4. Create eStream set: When the user selects this option, the builder creates an eStream set for all the installations done in the session. The Builder

maintains a current appId counter in the registry (not to be confused with the registry monitoring we are doing), and uses that to prompt the user with the appId to be used for this app set. The user can override that appId. The user is also prompted for a version number of the appInstallBlock. The output eStream set is created with the version number. Note: The Builder will NOT compress or encrypt any of the application files. This is done by the eStream server prior to transmitting the files to the client.

2) Builder UI to Install Monitor: Assuming the install monitor is a separate module, there will be an entry function to invoke the install monitor thus:

```
unsigned int InstallMonitor(
        IN  PUNICODE_STRING setup_name,  /* setup.exe path */
        IN PUNICODE_STRING dest_drive,    /* dest drive for install */
        IN PUNICODE_STRING file_for_file_info, /* file for storing file info */
        IN PUNICODE_STRING file_for_rel_info,  /* file for rel info */
        IN PUNICODE_STRING file_for_reg_info, /* file for storing reg info */
        IN BOOL append);              /* append to above 2 files if they exist */
```

<The file formats for the file info, rel file info and the reg info are yet to be fixed>.

3) Install Monitor to setup.exe of the application: The install monitor invokes the setup.exe using CreateProcess or similar Win32 API. Note that it has to invoke this as suspended, since we want to get the process_id and pass it on to the FSRFD so that the FSRFD can start monitoring all the requests for this process as well as its children. After that we want to resume the setup.exe process. (Alternatively, we can just pass the process id of the InstallMonitor process and the setup.exe would be the child of this process).

4) Install Monitor to FSRFD: The interface between these two is going to be very similar to the installmon interface in the prototype. The interface to FSRFD will support the following:

   a) MON_ACTIVATE: Start monitoring. Will pass the process-id, destination drive, system drive

```
DeviceIoControl(hDevice,      // handle to our FSRFD device
                MON_ACTIVATE,  // activate control code
                activateOptions,      // process-id, dest and sys drive
                sizeof(activateOptions),
                0, 0,
                &numBytesReturned,
                0);
```

b) MON_DEACTIVATE: Stop monitoring.

```
DeviceIoControl(hDevice,
        MON_DEACTIVATE,
        0, 0,
        0, 0,
        &numBytesReturned,
        0);
```

c) We would use event objects (using IoCreateNotificationEvent as in the installmon program) for the FSRFD to signal the Install Monitor about the availability of new data. There would be a single event object with the path "\\BaseNamedObjects\installmon" that the FSRFD will use to signal the Install Monitor that a new reg entry or file entry is available.

d) MON_GET_ENTRY: Get the next entry from the FSRFD. This would be either a file or registry update.

```
DeviceIoControl(m_deviceHandle,
                MON_GET_ENTRY,
                0, 0,
                monitorEntry,
                sizeof(monitorEntry),
                &numBytesReturned,
                0);
```

monitorEntry is a struct that is used to convey registry or file updates to the install monitor. The struct tentatively looks as follows:

```
struct MonitorEntry_t {
    UCHAR regOrFile;  // 'R' for registry 'F' for file
    UCHAR addOrDelete; // 'A' for add, 'D' for delete, 'U' for update
    UCHAR valueType; // only registry adds, value type
    ULONG nameLength; // length of name
    ULONG valueLength;   // name of value (when it is a string)
    UNICODE_CHAR  name[1];  // name of nameLength followed by value
                            // valueLength
};
```

Install Monitor Output contents

AppInstallBlock Contents

- **AibVersion**: Magic number or appInstallBlock version number (which identifies the version of the appInstallBlock structure rather than the contents).

- **AppId** (this may be a 64-bit number where the low 32 bits identify app's version number). AppId for Word on Win98 will be different from Word on WinNT (if it turns out that Word binaries are different between NT and 98).
- **VersionNo**: Version number. (this allows us to inform the client that the appInstallBlock has changed for a particular appId). Note: this is different from the low 32 bits of the appId. E.g. Word 97 and Word 98 will be differentiated using the low 32 bits of the appId. Or versions 1.0 and 2.0 of a software might be differentiated using the low 32 bits of the appId. However this field will be used when the Word 97 has been updated using a patch and the old binaries (for the same appId) are no longer available.
- **ClientOSBitMap**: Client OS supported bitmap or ID: for Win2K, Win98, WinNT and other future Oss we might support (it should be possible to say that this appInstallBlock is for more than one OS).
- <**ClientOSServicePack**: We might want to store the service pack level of the OS for which this appInstallBlock has been created. Note that when this field is set we cannot use multiple OS bits in the above field ClientOSBitMap>.
- **ISM**: Installation Set Map (ISM): contains
    - FileId (-1 denotes a deleted file)
    - Full path of the file where the file resides in eStream (e.g. Z:\...)

  Note: a range of fileIds will be reserved for eStream's own files. E.g. the appInstallBlock files could be referenced using these reserved fileIds.
- **FRM**: File Relocation Map (FRM): contains
    - FileId (same as the one in the ISM).
    - The path where the application expects it (e.g. C:\...)

  Note: The client should compare the FRM with the client machine. Any files that exist on the system (such as MSVCRT.DLL) will need to be copied to the client machine (instead of spoofing the file) and the client will need to be told to reboot the system.
- Registry spoof data: contains:

```
HKEY_CURRENT_USER
    Estream
            Registry Spoof
                    Add
                            HKCR
                                    ...
                            HKLM
                                    ...
                            HKCC
                                    ...
                    Remove
                            HKCR
                                    ...
                            HKLM
                                    ...
                            HKCC
```

- Initial profile data: The Builder will profile the application usage to get the sequences of the blocks accessed. The Builder will gather the access sequences sent by the OS to the file system. The profiler will stop when the profile noise exceed some tolerance level (ie. If X percentage of the pages of the application being profiled has been kicked out of memory). A list of entries where each entry has the following format:
  - o previous fileID
  - o previous blockID (blockID may be replaced with Offset and Length if variable sized cache blocks is supported)
  - o next fileID
  - o next blockID (blockID may be replaced with Offset and Length if variable sized cache blocks is supported)
  - o frequency.
- Initial pages for the app – The size of the initial prefetch blocks will be determined based on the minimum size of the data required to achieve good compression. A list of entries where each entry has the following format:
  - o FileID
  - o BlockID (blockID may be replaced with Offset and Length if variable sized cache blocks is supported).
- A comment field – this text might be used by the client to show the eStream app user any relevant info
- Special processing needed for this app – this could be a pointer to an EXE/DLL or the code itself in the appInstallBlock or a batch file.

Other Tools Related to the Builder

AppInstallBlock Editor/Modifier

This tool allows you to edit/modify one or all of :

a) the appId
b) the appInstallBlock version number
c) OS bit map (since we may need to add bits for other OSs once we know that binaries are the same for them).

of the appInstallBlock portion of the eStream set created by the builder.

AppInstall Compare Tool

This tool allows one to compare two eStream sets (both the appInstallBlocks and the actual files contained in the file set). Any differences are flagged to the user. This will be useful when we know that an application has the same set of binaries for multiple client OSs.

This tool will also be used to create appUpgradeBlock that will be used by the eStream client to seamlessly upgrade client's apps without necessarily destroying their config files. This is the only way to do it without needing an uninstall of the previous version.

Builder Tester

This program allows us to test the output of the Builder without requiring the eStream client/server set up.

eStream Builder Package Manager Low Level Design

Sanjay Pujare and David Lin

Version 0.1

[redacted]

Functionality

The eStream Application Builder Package Manager is responsible for packaging data gathered from the Installation Monitor, the Profile Manager, and the Upgrade Monitor into a set of data called the eStream Set. For the detail format of the eStream Set, see the separate document on eStream Set.

The Package Manager must perform the following task:

- Create the appInstallBlock containing C-File and Registry data from the Install Monitor; Prefetch data from the Profile Manager; and Updated C-File and Updated Registry data from the Upgrade Monitor
- Create a custom installation DLL needed by a specific applications and add to the appInstallBlock
- Create directory files associated with each directory of the application director and add metadata to the directory
- Create directory files associated with each Windows directory containing both the Spoofed files and Z-files
- Create Concatenated Application File (CAF) which is just a juxtaposition of the application files, eStream directory files, and AppInstallBlock
- Create Size Offset File Table (SOFT) which is a mapping of fileNumber to offset of the start of the CAF file
- Create Root Version Table (RVT) which is a mapping from the version of root to the file number of the root directory file
- Archive the CAF, SOFT, and RVT into a single structure called eStream Set suitable for uploading to the eStream Servers.

Data type definitions

The Package Manager doesn't have any internal data types. It must accept and understand data structures received from the Install Monitor and the Profile Manager. See Install Monitor and Profile Manager components for the description of the data structures.

The Install Monitor is responsible for generating the following list of information: list of copied-files, list of spoof-files, list of files with file numbers, list of add registry entries, and list of delete registry entries. The list of copied-files contains the files copied into non-application specific directories. The list of spoof-files consists of the files too large to be downloaded to the client in the AppInstallBlock. Those files are copied into some special directory on the Z drive for streaming. The list of files with file numbers consists of the files copied into the standard "Program Files" directory and the files that will be spoofed. The registry information is a list of registry key added or removed during the installation of the application.

```
Struct FileIndexTable {
   UINT NumEntries;
   Struct Entry {
      PUNICODE_STRING FilePathName;
      ULONG FileNumber;
   } Entries[NumEntries];
};
Struct FileCopied {
   UINT NumEntries;
   Struct Entry {
      PUNICODE_STRING FilePathName;
   } Entries[NumEntries];
}
Struct FileSpoofed {
   UINT NumEntries;
   Struct Entry {
      PUNICODE_STRING OldFilePathName;
      PUNICODE_STRING NewFilePathName;
   } Entries[NumEntries];
};
Struct RegistryInfo {
   UINT NumEntries;
   Struct Entry {
      PUNICODE_STRING KeyName;
      PUNICODE_STRING ValueName;
      PVALUE_DATA ValueData;
   } Entries[NumEntries];
};
Struct IniInfo {
   UINT NumFiles;
   Struct FileEntry {
      PUNICODE_STRING FilePathName;
      UINT NumSections;
      Struct SectionEntry {
         PUNICODE_STRING SectionName;
         UINT NumValues;
         Struct Entry {
            PUNICODE_STRING ValueName;
            PVALUE_DATA ValueData;
         } Entries[NumValues];
      } Entries[NumSections];
```

```
    } Entries[NumFiles];
};
```

The Profile Manager generates AccessCounts and the PrefetchBlocks data with the structures shown below.

```
Struct AccessCounts {
   UINT NumEntries;
   Struct Entry {
      PUNICODE_STRING FilePathName;
      ULONG Frequency;
   } Entries[NumEntries];
};
Struct PrefetchBlocks {
   UINT NumEntries;
   Struct Entry {
      PUNICODE_STRING FilePathName;
      ULONG BlockNumber;
   } Entries[NumEntries];
};
```

The eStream Set has the following data structure (described in more detail in the separate eStream Set document):

```
Struct eStreamSet {
   Struct eStreamSetHeader header;
   Struct eStreamSetRVT rvt;
   Struct eStreamSetSOFT soft;
   Struct eStreamSetCAF caf;
};
```

Interface definitions

Function 1 : CreateEStreamSet

```
// Create the initial eStream Set from the data
// retrieved from the Install Monitor and the
// Profile Manager.
// This function is called only by the Builder
// UI after data is obtained from Install
// Monitor and Profile Manager.
int CreateEStreamSet(
   IN PFILE_INDEX_TABLE FIT,
   IN PFILE_SPOOFED SpoofFiles,
   IN PFILE_COPIED CopiedFiles,
   IN PREGISTRY_INFO AddRegistry,
   IN PREGISTRY_INFO RemoveRegistry,
   IN PINI_INFO IniInfo,
   IN PACCESS_COUNTS AccessCounts,
   IN PPREFETCH_BLOCKS PrefetchBlocks,
    IN PVOID DllCode,
    IN PUNICODE_STRING Comment,
    OUT PESTREAM_SET EstreamSet)

Input:
    FIT: File Index Tree contains the file
       number of the directories, spoofed
       files, and standard files

    CopiedFiles: pointer to a list of files
       To be copied to AppInstallBlock

    SpoofFiles: pointer to a list of files
       To be spoofed on the client

    AddRegistry: pointer to a list of registry
       Data to add
    RemoveRegistry: pointer to a list of
       Registry data to remove
    IniInfo: pointer to a list of ini changes

    AccessCounts: pointer to the list of
       Files with the access frequency

    PrefetchBlocks: pointer to the prefetch data
       To be inserted into the appInstallBlock
       Of the eStream Set

    DllCode: pointer to DLL Code

    Comment: pointer to comment string

Output:
    EstreamSet: pointer to the eStream Set

Return Value:
    Success or failure of the packaging process

Comments:
    The eStream Set will be large for most
    application.  Intermediate data will be
    stored on the local hard-drive.

Errors:
    OutOfStorage: failure to find enough storage
       For this eStream Set
```

```
FileNotFound: failure to find the files
  Specified by either ListCFiles or
  ListZFiles
```

Function 2 : UpgradeEStreamSet

```
// Upgrade the eStream Set to the latest
// version. This function is only called by
// the Upgrade Manager within the same process.

int UpgradeEStreamSet(
   INOUT PESTREAM_SET EstreamSet,
   IN PFILE_INDEX_TABLE UpgFIT,
   IN PFILE_SPOOFED UpgSpoofFiles,
   IN PFILE_COPIED UpgCopiedFiles,
   IN PREGISTRY_INFO UpgAddRegistry,
   IN PREGISTRY_INFO UpgRemoveRegistry,
   IN PACCESS_COUNTS UpgAccessCounts,
   IN PPREFETCH_BLOCKS UpgPrefetchBlocks,
   IN PVOID UpgDllCode,
   IN PUNICODE_STRING UpgComment)

Input:
   UpgFIT: File Index Tree contains the file
     number of the directories, spoofed
     files, and standard files

   UpgCopiedFiles: pointer to a list of files
     To be copied to AppInstallBlock

   UpgSpoofFiles: pointer to a list of files
     To be spoofed on the client

   UpgAddRegistry: pointer to a list of
     Registry data to add

   UpgRemoveRegistry: pointer to a list of
     Registry data to remove

   UpgAccessCounts: pointer to the list of
     Files with the access frequency

   UpgPrefetchBlocks: pointer to the prefetch
     Data to be inserted into the
     AppInstallBlock Of the eStream Set

   UpgDllCode: pointer to DLL Code
   UpgComment: pointer to comment string

Output:
   EstreamSet: pointer to the eStream Set

Return Value:
   Success or failure of the packaging process

Comments:
   The eStream Set will be large for most
   application.  Intermediate data will be
   stored on the local hard-drive.

Errors:
   OutOfStorage: failure to find enough storage
      For this eStream Set

   FileNotFound: failure to find the files
      Specified by either ListCFiles or
      ListZFiles
```

Function 3 : InsertProfileData

```
// Insert profile and prefetch data into the
// eStream Set. This function is only called by
// the Merge Manager within the same process.

int InsertProfileData(
   INOUT PESTREAM_SET EstreamSet,
   IN PACCESS_COUNTS AccessCounts,
   IN PPREFETCH_BLOCKS PrefetchBlocks)

Input:
   EstreamSet: pointer to old eStream Set
      Before the insertion of the profile
      Data

   AccessCounts: pointer to the list of
      Files with the access frequency

   PrefetchBlocks: pointer to the prefetch data
      To be inserted into the appInstallBlock
      Of the eStream Set

Output:
   EstreamSet: pointer to the new eStream Set

Return Value:
   Success or failure of the insertion process

Comments:
   The eStream Set will be large for most
   application.  Intermediate data will be
   stored on the local hard-drive.

Errors:
   OutOfStorage: failure to find enough storage
      For this eStream Set

   FileNotFound: failure to find the files
      Associated with the prefetch blocks
```

Component design

The pseudo-code for the function *CreateEStreamSet* is described below:

```
{
   Create AppInstallBlock (AIB) from the following input files:
         o  SpoofFiles
         o  CopiedFiles
         o  AddRegistry
         o  RemoveRegistry
         o  Prefetch
         o  Comment
         o  DLLcode
   Assign AppInstallBlock with a unique fileNumber given by the IM;
   Record Root fileNumber in the first entry of Root fileNumber Table (RFT);
   Move AppInstallBlock under the Root directory by adding a new entry in the
      Directory structure;
   Create a Concatenation Application File (CAF) header;
   Create a Size Offset File Table (SOFT) header;
   For each (file in FIT) {
      If (file is a directory) {
         Create the directory with new list of fileNumber, filename, and
            Metadata;
      } Else {
         Find the file in the proper location on the HD;
      }
      Append the file or directory to the end of the CAF file;
```

```
            Append the fileNumber, offset into CAF, and size of file in SOFT;
        }
        Archive CAF, SOFT, and RFT into a single eStream Set;
        Return eStream Set;
    }
```

The pseudo-code for the function *UpgradeEStreamSet* is mentioned below:

```
    {
        Extract previous version PrevAppInstallBlock from eStream Set;
        Create new AppInstallBlock with new FileNumber;

        Extract PrevSpoofFiles and PrevCopiedFiles from PrevAppInstallBlock;
        Divide the C-Files into SpoofFiles and CopiedFiles;
        Add PrevSpoofFiles to SpoofFiles;
        Add PrevCopiedFiles to CopiedFiles;

        Extract PrevAddRegistry and PrevRemoveRegistry data from
            PrevAppInstallBlock;
        Add any unique ((UpgAddRegistry plus PrevAddRegistry) minus
            UpgRemoveRegistry) in the new AppInstallBlock AddRegistry section;
        Add any unique ((UpgRemoveRegistry plus PrevRemoveRegistry) minus
            UpgAddRegistry in the new AppInstallBlock;

        Add UpgPrefetch data to new AppInstallBlock;
        Add UpgDllCode data to new AppInstallBlock;
        Add UpgComment data to new AppInstallBlock;

        For each (directory in UpgFIT) {
            If (any child fileNumber has changed) {
                Create new directory with updated fileNumber;
                Append file to end of Concatination Application File (CAF);
                Append Size Offset File Table (SOFT) with new entry;
            }
        }
        Append new AppInstallBlock to the end of CAF file;

        Prepend Root FileNumber Table (RFT) with new Root entry;
    Archive CAF, SOFT, and RFT into a single eStream Set;
    Return eStream Set;
}
```

The pseudo-code for the function *InsertProfileData* is mentioned below:

```
{
    // not needed unless merging of uploaded profile data is supported
}
```

Testing design

This document must have a discussion of how the component is to be tested.

- **Unit testing plans**

  The plan for unit testing Package Manager includes the development of a driver program. This driver interfaces to the Package Manager and invokes the functions with different parameters. The list of possible cases is described below:

  1. Test all interfaces by driving the input parameters with different type of add and remove registry values.
  2. Test all interfaces by driving the input parameters by varying numbers of spoof and copied files.
  3. Test all interfaces by driving the input parameters with some prefetch information.
  4. Test all interfaces for meaningless input values from the IM and PM.
     - Prefetch block containing file number not assigned by IM.
     - IM assigning non-continguous file numbers.
  5. Test upgrade interface for capability to detect and handle bad eStream Set gracefully.
  6. Test upgrade interface and make sure it can detect overlapping file number assignments.
  7. Test upgrade interface and make sure prefetch blocks are not referencing old file number from previous versions.

- **Stress testing plans**

- **Coverage testing plans**

- **Cross-component testing plans**

  The output data from the Package Manager is called the eStream Set. This eStream Set is the input to a stand-alone test program called the *eStream Extrac-*

tor. The Extractor unpacks and 'install' the eStream Set into the local machine without an eStream client file system installed. This test is used to quickly verify that the eStream Set can be run on a pristine machine. Some of the possible variations of the Extractor test includes:

1. Non-default system variable names. I.e. %SystemRoot% set to "D:\Win" instead of "C:\Winnt".
2. Non-default eStream FS drive letter. Use Y instead of Z.

Upgrading/Supportability/Deployment design

The Package Manager logs all error messages to a predefined file common to all components of the Builder program. Every Builder component prints the error message along with its component name. This allows the user of the Builder program to quickly track down any problem during the Building of a new eStream Set.

Open Issues

- Which Builder component creates the installation DLL when the application needs the custom installation code? Is a new component needed to create the custom DLL separately and insert into AppInstallBlock in the eStream Set as needed?

eStream Builder Profile Manager Low Level Design

Sanjay Pujare and David Lin
Version 0.2
[redacted]

Functionality

The eStream Application Builder Profile Manager is responsible for the following:

- Receive request from the UI Component for one or more application executables to profile.
- Accumulate each run of the profile data in a data structure suitable for merging.
- Invoke each application executable for a fixed amount of time, for a fixed number of prefetch blocks, for a simple start-stop of the program, or for multi-level profiling based on scripts or manual usage of an application.
- Communicate with File Access Monitor (FAM) kernel-mode driver using IOCTLs to start and stop profiling.
- Obtain the complete file access sequence data from the FAM.
- Process the file access sequence into two parts: a table of file access frequency and a list of prefetch blocks.
- Send the resulting profile data to Package Manager component for integration into the AppInstallBlock.

This component will probably exist as a class object and will be instantiated by the Builder user interface component. The component will run in the same process as the user interface component. Please see Builder User Interface component document for more information on that component.

Data type definitions

The Profile Manager imports the file access sequence from the FAM. The data structure is described below: (Please see the File Access Monitor for detail information on the meaning of each field in the data structure)

```
Struct SequenceData
{
  UINT NumEntries;
  Struct Entry
  {
    UNICODE_STRING FilePathName;
    BOOL IsAccessingMetaData;
    ULONG Offset;
    ULONG Size;
  } Entries[NumEntries];
```

```
        };
```

Profile Manager creates two data structure from the data received from the Install Monitor and the FAM. The consumer of the output data structure is the Package Manager. These data structures is described in the following two structures:

```
        Struct PrefetchBlockList {
           UINT NumSections;
           Struct PrefetchBlocks {
              UINT BlockType;
              UINT NumEntries;
              Struct Entry {
                 UNICODE_STRING FilePathName;
                 ULONG BlockNumber;
              } Entries[NumEntries];
           } Entries[NumSections];
        };
        Struct ProfileApplications {
           UINT NumEntries;
           Struct Entry {
              UNICODE_STRING FilePathName;
              UNICODE_STRING Arguments;
           } Entries[NumEntries];
        };
```

The access count is used to order the list of files in a directory according to metadata file access frequency. The prefetch data is encorporated into the AppInstallBlock by the Package Manager to be used by the eStream Client.

Interface definitions

Function 1 : StartProfiling

```
    int StartProfiling(
       IN PPROFILE_APPLICATIONS AppList,
       IN UINT Type,
       IN UINT TypeData,
       OUT PPREFETCH_BLOCKS PrefetchBlocks)

    Input:
       AppList: a list of file pathnames and
          Arguments to run the application.

       Type: type of profiling to do
          SIMPLE: start and stop application
          TIMEBASED: profile for a fixed time and
          terminate application
        SIZEBASED: profile for a fixed size of
          Profile data

      TypeData: extra data for profile type
        If ProfileType==SIMPLE, TypeData is
          Ignored
        If ProfileType==TIMEBASED, TypeData is
          Length of time in seconds
        If ProfileType==SIZEBASED, TypeData is
          Size of profile data in bytes
        If ProfileType==COMMANDBASED, TypeData is
          Pointer to a possible list of script
          Files to be invoked

      PrefetchBlocks: List of prefetch blocks
        To add to the AppInstallBlock

    Return value:
      Success or failure code of the profiling

    Comments:
      The profile manager component actually send
      IOCTLs to the file filter device driver to
      Start and stop the gathering of the profile
      Data and to retrieve the profile data.

    Errors:
      FileNotFound: some of the application
        Executables in the list may not exist or
        Not readable
      ProfileTimeout: failed to gather the desired
        Size of the profile data after certain
        Amount of time
      DriverFailure: File Access Monitor return
        Failure code to the Profile Manager which
        Must propagate the error to the user
        Interface
```

Component design

Application Invocation

To start profiling, the component must have a list of application pathname to be invoked. The pathname may be an executable with a list of arguments. Or the pathname may be a Windows short-cut file. If the pathname is an exe file, then the standard Win32 API *CreateProcess()* is used. On the other hand, if the file is a Windows short-cut file, then the component needs to extract relevant information from the short-cut file to make the proper *CreateProcess()* invocation. The following is a pseudo-code for extracting path-name and argument information from the short-cut file using IShellLink interface:

```
GetInfoFromShortCutFile(char *strPath, char *strArg)
{
    IshellLink *psl;
    WIN32_FIND_DATA fd;
    if (SUCCEEDED( CoCreateIstance(CLSID_ShellLink,
                   NULL,
                   CLSCTX_INPROC_SERVER,
                   IID_IShellLink,
                   (LPVOID*) &psl)))
    {
        psl->GetPath(strPath, MAX_LEN, &fd, 0);
        Psl->GetArguments(strArg, MAX_LEN);
        psl->Release();
    }
}
```

Command-based Profiling
One of the options for profiling include the ability to identify blocks of files accessed when specific application command is invoked. The profiler prompts the user for the desired actions (ie. Open document, save document, etc) and gathers file blocks that are accessed corresponding to those actions. These commands are saved into the AppInstall-Block for eStream client to intelligently pick the proper set of blocks to stream. The following is an enumeration of some divisions of prefetch blocks:

- Start Application
- End Application
- Save Document
- Open Document
- Cut Sections
- Copy Sections
- Paste Sections

Communication with kernel-mode driver (FAM)
The Profile Manager communicates with the kernel-mode driver to retrieve the information on the blocks of files accessed. The profile manager waits for a signal from the FAM indicating a new data is available for retrieval. FAM also signals the profile manager when the profiled application terminates. FAM uses *KeSetEvent()* to send a 'data available' event signal to the profile manager. Profile manager calls *KeWait-ForSingleEvent()* or *KeWaitForMultipleEvent()* to wait for a signal from the kernal-mode driver. *KeClearEvent()* is called by the FAM when the signal to profile manager should be deactivated.

Pseudo-code
The pseudo-code for the function *StartProfiling* is described below:

```
{
    Initialize GlobalFileAccessCounts and GlobalPrefetchBlocks;
    Load FAM if not loaded;
    For each (executable in the list of application) {
        Start FAM by sending it process ID of the Profile Manager;
        Create new process for this executable and run it;
        Switch (Type of Profiling) {
        Case SIMPLE:
            Loop {
                Wait for an event from FAM;
                Get Status from FAM;
                Get SequenceData from FAM;
            } Until application start up;
            Break;
        Case TIMEBASED:
            Loop {
                Wait for an event from FAM;
                Get Status from FAM;
                Get SequenceData from FAM;
            } Until fixed time unit;
            Break;
        Case SIZEBASED:
            Loop {
                Wait for an event from FAM;
                Get Status from FAM;
                Get SequenceData from FAM;
            } while (size < fixed amount);
            Break;
        Case COMMANDBASED:
            For each command in the list {
                If (script exist)
                    Run script on the executable program;
                Else
                    Prompt operator for proper action;
            Loop {
                Wait for an event from FAM;
                Get Status from FAM;
                Get SequenceData from FAM;
            } Until script completed;
        }
    }

    Send WM_QUIT message to the application process;
    Loop {
        Wait for an event from FAM;
        Get Status from FAM;
        Get SequenceData from FAM;
    } Until application quit;
    Inform FAM to stop gathering profile data;

    Compute PrefetchBlocks from the SequenceData and append to
        GlobalPrefetchBlocks;
  }
  Unload File Access Monitor (if possible);
  Return GlobalPrefetchBlocks;
}
```

Testing design

- **Unit testing plans**

The plan for unit testing the Profile Manager includes developing a driver to connect to the interface between the Profile Manager and the Builder UI. The driver conducts the following types of tests:

1. Test different type of profiling including simple profiling, time-based profiling, size-based profiling, and script-based profiling.
2. Test different executable programs and make sure the output data "makes sense".
3. Test a list of executables for merging capability of the Profile Manager.
4. Test the interface between Profile Manager (PM) and the File Access Monitor (FAM) using FAM as the test driver. The FAM can check for any valid IOCTL calls from the PM. FAM can also reply to IOCTL calls with different values in the IRP to simulate all possible cases.

- **Stress testing plans**

- **Coverage testing plans**

- **Cross-component testing plans**

  Cross-component testing for the Builder program is described in the Package Manager low-level design document.

Upgrading/Supportability/Deployment design

The Profile Manager logs all error messages to a predefined file common to all components of the Builder program. Every Builder component prints the error message along with its component name. This allows the user of the Builder program to quickly track down any problem during the Building of a new eStream Set.

Open Issues

- How to automate profiling so the application doesn't require any user intervention? Look into using Rational TestSuite programs.
- Can a subset of Winstone be used for profiling? How do we determine which part of the profile data is more useful to the end-user?
- Should Profile Manager actually create data structures like PrefetchBlocks which require FileNumber assignments? Or should Profile Manager just create the output data without knowning FileNumbers? Then Package Manager associates the file numbers assigned by the Install Monitor with the profile data gathered by the Profiler.

Tricky Builder Issues

Author: Sanjay Pujare

This document enumerates all those tricky issues that may make the Builder's job difficult. Even though some solution may be proposed for some issues, not every issue would have a solution described in this document. The purpose of this document is mainly to keep track of Builder issues that may impose some limitations on the eStream technology. This way Omnishift marketing and deployment are aware of these limitations.

1) The Builder cannot capture updates to existing files in an intelligent fashion (i.e. if the updates are based on a context or existing contents, it is very difficult to capture that). So the current Builder will just flag an error, if such an update occurs.

   <u>Solution</u>
   - ❑ These updates are probably very rare, so we can defer it to the next release.
   - ❑ For this release, we can try to solve this on a case by case basis e.g. we will try to solve this issue for INI files.
   - ❑ Based on our understanding of general app installations, we might be able to make some generalizations that we can use in eStream e.g. Only certain files get updated; there is a definite pattern of updates.
2) The current Builder drivers are based on the NT driver model, and hopefully we can implement the same functionality in the Win98 drivers, but this needs to be ensured. (This shouldn't be an issue, but...)
3) We need to think some more about those cases when device drivers are installed by apps. Issues that can arise:
   - ❑ This may not work correctly on the eStream client, just because the driver installation didn't take place properly.
   - ❑ The Builder would need to be able to figure out in an automated way if a client reboot is required or not.
   - ❑ If the driver installation is h/w or s/w specific that can be difficult to tackle.

   <u>Solution</u>
   - ❑ As we eStream more apps and gain more experience, we should be able to figure out solutions.
4) There could be an ambiguity when the Installmon is trying to change absolute paths (or absolute values in general) to relative paths. e.g. A path like C:\WINNT can be changed to %SystemRoot% or %windir% since both of those environment variables are set to "C:\WINNT" on my system.

   <u>Solution</u>
   - ❑ We can prioritize env vars and registry keys as described in the BuilderUI-LLD design document.

- We should encourage Builder operators to use as distinct values as possible for env variables and registry keys for the Builder machine.

eStream BuilderUI Low Level Design

Sanjay M Pujare
<Date>

Functionality

The BuilderUI is the user interface part of the Builder. The operator uses this interface to use various functions provided by the Builder. Note that this UI may or may not be a graphical user interface. This low-level design is based on the assumption that a graphical user interface is not necessary.

Data type definitions

Interface definitions

Component Design

When the Builder UI is invoked with command line arguments which indicate that this was invoked by the `Runonce` mechanism of Windows, the control is transferred to the `InstallMon::startCaptureAfterReboot()` function with the command line arguments passed as arguments to the function. When the Builder is invoked normally, it presents a menu which is managed by the function `MainMenu`.

MainMenu

```
This function manages the following menu hierarchy. Each
menu option (leaf node) is followed by a function name in
parentheses that is called to handle the option.
1) eStream Set Menu
   1) New eStream Set (NewEStreamSet)
   2) Open eStream Set (OpenEStreamSet)
   3) Save New/Upgraded eStream Set (EstreamSetCreation)
2) Monitoring Menu
   1) Start Monitor (StartMonitor)
   2) Stop Monitor (StopMonitor)
   3) Check Status (CheckStatus)
   4) Inform Machine Reboot (InformMachineReboot)
   5) Get and Resolve Registry Set (GetRegistrySet)
   6) Get and Resolve Files Set (GetFilesSet)
3) Profiling Menu
   1) Set the location of app executable (GetAppPath)
   2) Gather Initial Profile (GatherInitialProfile)
4) eStream Set Creation Menu
   1) Set custom DLL (GetCustomDLL)
   2) Set User Comment (GetUserComment)
   3) Set environment variables (GetEnvVars)
```

```
  4) Set Reboot flag (GetRebootFlag).
  5) Set License Agreement (GetLicenseAgreement).
```

NewEStreamSet

```
{
  If there is an existing eStream set that hasn't been
  saved, warn the user.
  Get the following values from the user:
    • Name of the app setup program in gAppSetup
    • Dest directory where app will be installed in gDest-
      Dir (provide a default value).
    • Dest location to store the new eStream set in
      gDestEstreamPath (provide a default value).
}
```

OpenEStreamSet

```
{
  If there is an existing eStream set that hasn't been
  saved, warn the user.
  Get the following values from the user:
  • Location of the existing eStream set in gSrcEstream-
    Path (provide a default value).
  • Whether the user wants to create an upgrade from this,
    or just wants to change the existing eStream set (gUp-
    grade)
  • If this is an upgrade (gUpgrade is true), get all the
    values obtained by NewEstreamSet (i.e. gAppSetup,
    gDestDir, gDestEstreamPath).

  Read the eStream set pointed to by gSrcEstreamPath;
  Load the existing file tables in gSrcCopiedFiles,
  gSrcSpoofedFiles and gSrcEFSFiles arrays;
}
```

EstreamSetCreation

```
{
  If there is no working eStream set, give error and re-
  turn.
  if (gUpgrade) {
    Call UpgradeEStreamSet() with appropriate arguments;
  }
  else {
    Call CreateEStreamSet() with appropriate arguments;
  }
  if (gDestEstreamPath is not set) {
    assert(this is an update of an existing eStream set
      and not an upgrade or a new eStream set creation);
    gDestEstreamPath = gSrcEstreamPath;
  }
  Save the eStream Set from memory to file to
    gDestEstreamPath;
}
```

StartMonitor

```
{
  If there is no working eStream set, give error and re-
  turn.
  if (gUpgrade) {
    Combine the gSrcSpoofedFiles and gSrcEFSFiles into an
    array fileTableArray as expected by startCapture
    below;
  }
  Call InstallMon::startCapture(gAppSetup, gDestDir,
       gUpgrade, fileTableArray);
}
```

StopMonitor

```
{
  If monitoring wasn't started, give error and return.
  Call InstallMon::stopCapture();
}
```

CheckStatus

```
{
  Ensure that monitoring was started and not stopped
  Call InstallMon::checkSetupStatus();
}
```

InformMachineReboot

```
{
  Ensure that we are in the middle of monitoring.
  Call InstallMon::machineToBeRebooted();
}
```

GetRegistrySet

```
{
  Call InstallMon::getRegistryList();
  Store the set in gNewRegistry data structure;
}
```

GetFilesSet

```
{
  Call InstallMon::getFilesList();
```

```
    Store the set in gNewFiles data structure;
    Also we need to capture changes made to INI files; since
    this has not been taken care of in the app install block
    and other parts of the Builder+Client we will need to
    make changes in all those components which are affected.
}
```

GetAppPath

```
{
    Ensure that all the InstallMon related data was captured.
    Get the location of the app that needs to be run
    to gather profiling info;
    Store it in gProfileAppPath;
}
```

GatherInitialProfile

```
{
    // this is the function that is used to get the initial
    // profile data (i.e. set of pages prefetched when an
    // eStream app is started for the first time)
    // implementation of this is yet to be defined
}
```

GetCustomDLL

```
{
    Get the location of the custom DLL file, validate that it
    is a DLL and store the path in gCustomDLLPath;
}
```

GetUserComment

```
{
    Get the user comment (optionally by browsing a text file)
    and store it in gUserComment;
}
```

GetEnvVars

```
{
    Call the InstallMon::setEnvVars() function;
}
```

GetRebootFlag

```
{
    Until we come up with an algorithm to determine if a re-
    boot is required for an eStream app, get this value from
    the user. The default is FALSE: we do not want to reboot
    the client PC when the user subscribes to this eStream
    app.
}
```

<u>GetLicenseAgreement</u>

```
{
  Get license agreement from the user. This could be:
    ❑ either, a default OmniShift license agreement
    ❑ or, a default ASP agreement
    ❑ or, license agreement that the app displayed.
  Let the user decide and enter the proper one. Provide a
  default based on our policy.
}
```

Interesting issues to deal with:

Testing design

Unit testing plans

Testing of the UI itself is a comparatively trivial task. The testing will basically consist of traversing the whole menu hierarchy. Since the menu is similar to a typical File Open -> File Edit -> File Save kind of a user application, this can be tested using simple hooks.

Stress testing plans

Since the Builder will be used in house at least initially only simple stress testing should be necessary. Make sure that Builder doesn't crash in the middle of processing so that we don't lose important data. Performance is not considered to be important.

Coverage testing plans

Basically the following 3 paths will be exercised:

1. Create a new eStream set
2. Open an existing eStream set to modify some data in it
3. Open an existing eStream set to create an upgrade for it

Cross-component testing plans

Will be tested as a component of the whole builder.

Upgrading/Supportability/Deployment design

Deployment: This will be used in-house, so no deployment considerations.

Open Issues

- We need to think about the problem of converting absolute file paths discovered in the monitoring process to paths relative to some application or system registry key. Although most cases may not present a problem, we may have some difficult cases, which may make this problem non-automatable i.e. we may need some user intervention. Consider the case:

  KEYONE = C:\FOO\BAR

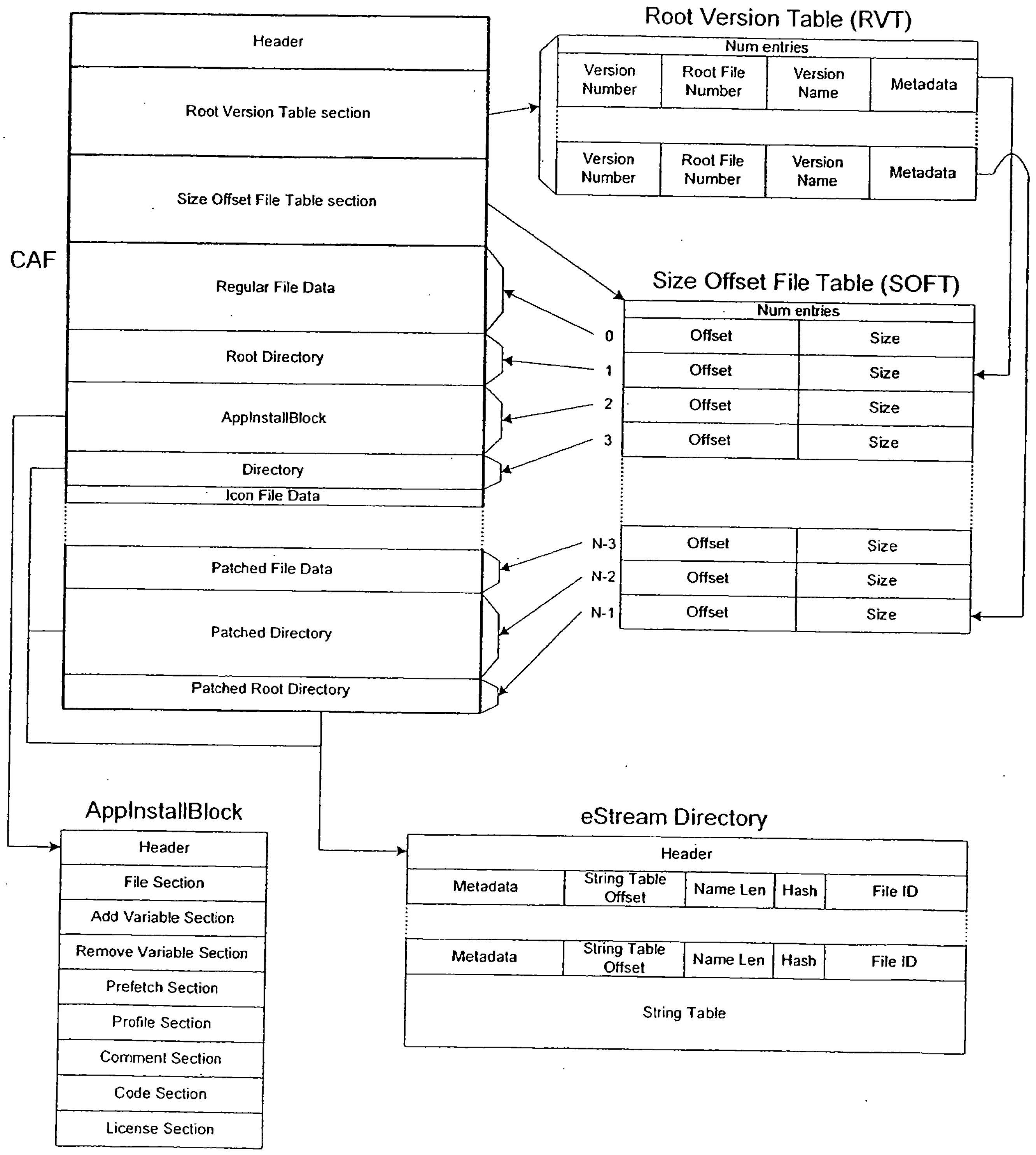
  KEYTWO = C:\FOO

  KEYTHREE = BAR

  If we notice that a file was copied to C:\FOO\BAR it won't be possible to convert this absolute path to a unique relative path since there are 2 solutions possible: %KEYONE% or %KEYTWO%\%KEYTHREE%.

  The way to solve this is by tracking only a set of well-known environment variables and registry keys. Also in this set we prioritize all of them. So in the above case, %KEYONE% will be preferred over %KEYONE%\%KEYTHREE% just because of the way they were prioritized.

Format of the eStream Set



v 0.2

eStream Set Format Low Level Design

Sanjay Pujare and David Lin
Version 0.3
[redacted]

Functionality

The eStream Set is a data set associated with an application suitable for streaming over the network. The eStream Set is generated by the eStream Builder program. This program converts locally installable applications into the eStream Set. This document describes the format of the eStream Set.

Note: Fields greater than a single byte is stored in little-endian format. All strings are in Unicode unless specifically stated otherwise. The eStream Set file size is limited to 2^64 bytes.

Data type definitions

The format of the eStream Set consists of 4 sections: header, Root Version Table (RVT), Size Offset File Table (SOFT), and Concatenation Application File (CAF) sections.

1. Header section

- **MagicNumber [4 bytes]:** Magic number identifying the file content with the eStream Set
- **ESSVersion [4 bytes]:** Version number of the eStream Set format.
- **AppID [16 bytes]:** A unique application ID for this application. This field must match the AppID located in the AppInstallBlock. Guidgen is used to create this identifier.

- **RVToffset [8 bytes]:** Byte offset into the start of the RVT section.
- **RVTsize [8 bytes]:** Byte size of the RVT section.
- **SOFToffset [8 bytes]:** Byte offset into the start of the SOFT section.
- **SOFTsize [8 bytes]:** Byte size of the SOFT section.
- **CAFoffset [8 bytes]:** Byte offset into the start of the CAF section.
- **CAFsize [8 bytes]:** Byte size of the CAF section.

- **VendorNameLength [2 bytes]:** Byte length of the vendor name.
- **VendorName [X bytes]:** Name of the software vendor who created this application. I.e. "Microsoft". Null-terminated.
- **AppBaseNameLength [2 bytes]:** Byte length of the application base name.
- **AppBaseName [X bytes]:** Base name of the application. I.e. "Word 2000". Null-terminated.

- **MessageLength [2 bytes]:** Byte length of the message text.
- **Message [X bytes]:** Message text. Null-terminated.

2. Root Version Table (RVT) section

The Root version entries are ordered in a decreasing value according to their file numbers. The Builder generates unique file numbers within each eStream Set in a monotonically increasing value. So larger root file number implies later versions of the same application. The latest root version is located at the top of the section to allow the eStream Server easy access to the data associated with the latest root version.

- **NumberEntries [4 bytes]:** Number of patch versions contained in this eStream Set. The number indicates the number of entries in the Root Version Table (RVT).

Root Version structure: (variable number of entries)

- **VersionNumber [4 bytes]:** Version number of the root directory.
- **FileNumber [4 bytes]:** File number of the root directory.
- **VersionName [32 bytes]:** Application version name. I.e. "SP 1".
- **Metadata [32 bytes]:** See eStream FS Directory for format of the metadata.

3. Size Offset File Table (SOFT) section

The SOFT table contains information to locate specific files in the CAF section. The entries are ordered according to the file number starting from 0 to NumberFiles-1.

- **NumberFiles [4 bytes]:** Number of entries in this section.

SOFT entry structure: (variable number of entries)

- **Offset [8 bytes]:** Byte offset into CAF of the start of this file.
- **Size [8 bytes]:** Byte size of this file. The file is located from address Offset to Offset+Size.

4. Concatenation Application File (CAF) section

CAF is a concatenation of all file or directory data into a single data structure. Each piece of data can be a regular file, an AppInstallBlock, an eStream FS directory file, or an icon file.

a. *Regular Files*

- **FileData [X bytes]:** Content of a regular file

b. *AppInstallBlock (See AppInstallBlock document for detail format)*

A simplified description of the AppInstallBlock is listed here. For exact detail of the individual fields in the AppInstallBlock, please see AppInstallBlock Low-Level Design document.

- **Header section [X bytes]:** Header for AppInstallBlock containing information to identify this AppInstallBlock.
- **Files section [X bytes]:** Section containing file to be copied or spoofed.
- **AddVariable section [X bytes]:** Section containing system variables to be added.
- **RemoveVariable section [X bytes]:** Section containing system variables to be removed.
- **Prefetch section [X bytes]:** Section containing pointers to files to be prefetched to the client.
- **Profile section [X bytes]:** Section containing profile data. (not used in eStream 1.0)
- **Comment section [X bytes]:** Section containing comments about AppInstallBlock.
- **Code section [X bytes]:** Section containing application-specific code needed to prepare local machine for streaming this application
- **LicenseAgreement section [X bytes]:** Section containing licensing agreement message.

c. *EStream Directory*

An eStream Directory contains information about the subdirectories and files located within this directory. The information includes file number, names, and metadata associated with the files.

- **MagicNumber [4 bytes]:** Magic number for eStream directory file.
- **StringTable [4 bytes]:** Byte size offset to beginning of the string table.
- **StringTableLength [4 bytes]:** Byte size length of the string table.
- **ParentFileID [16+4 bytes]:** AppID+FileNumber of the parent directory. AppID is set to 0 if the directory is the root.
- **SelfFileID [16+4 bytes]:** AppID+FileNumber of this directory.
- **NumFiles [4 bytes]:** Number of files in the directory.

Fixed length entry for each file in the directory:

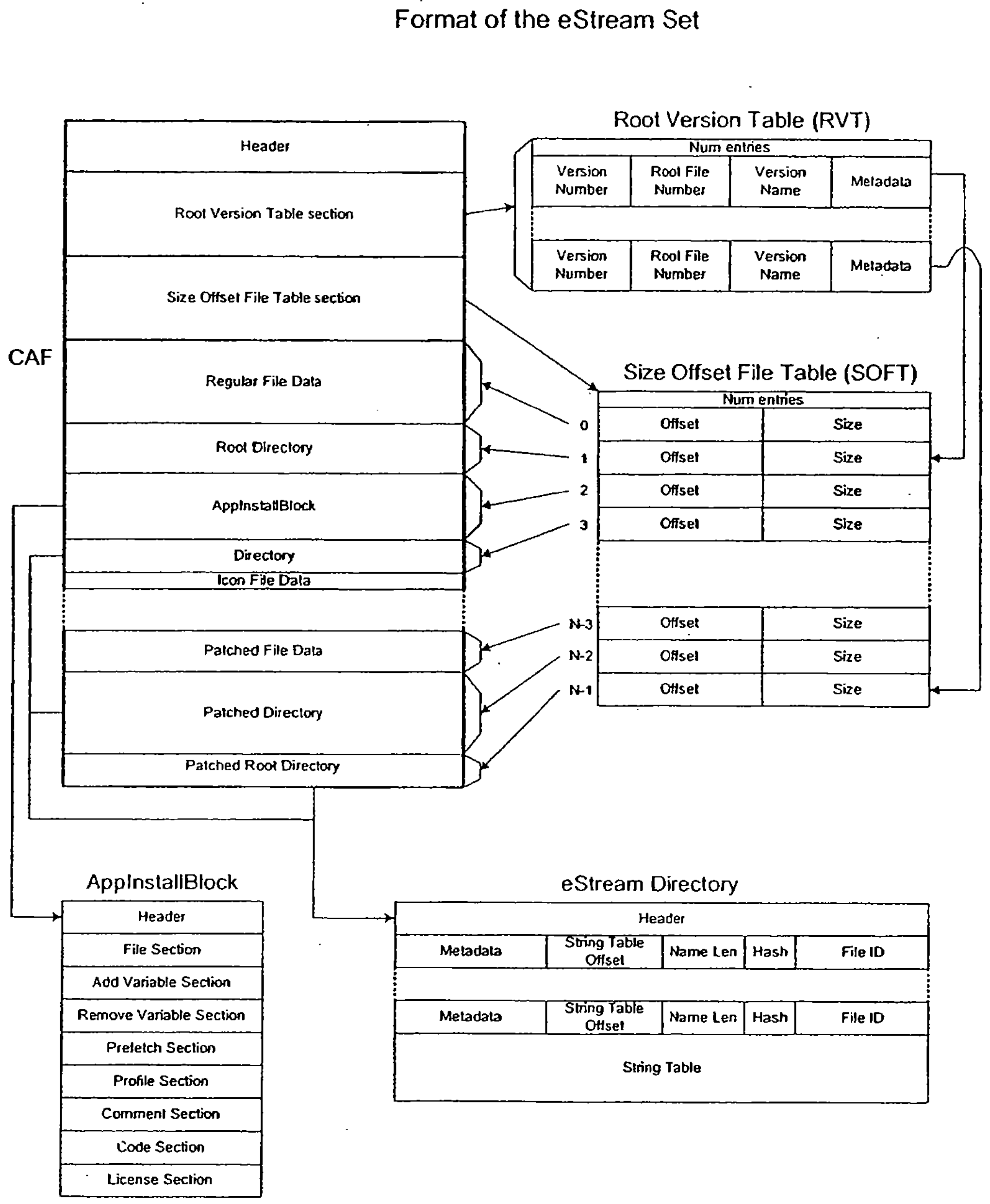
- **FileID [16+4 bytes]:** AppID+FileNumber of each file in this directory.
- **NameHash [4 bytes]:** Hash value of the file name. Algorithm TBD.

- **FileNameOffset [4 bytes]:** Offset where the file name is located, relative to the beginning of the string table.
- **FileNameLength [4 bytes]:** Byte size length of the file name that is null-terminated.
- **Metadata [32 bytes]:** The metadata consists of file **byte size** (8 bytes), file **creation time** (8 bytes), file **modified time** (8 bytes), **attribute flags** (4 bytes), **eStream flags** (4 bytes). The bits of the **attribute flags** have the following meaning:
  - **Bit 0:** Read-only – Set if file is read-only
  - **Bit 1:** Hidden – Set if file is hidden from user

  The bits of the **eStream flags** have the following meaning:
  - **Bit 0:** ForceUpgrade – Used only on root file. Set if client is forced to upgrade to this particular version if the current root version on the client is older.
  - **Bit 1:** RequireAccessToken – Set if file require access token before client can read it.
  - **Bit 2:** IsDirectory – Set if the file is a eStream Directory.

d. Icon files

- **IconFileData [X bytes]:** Content of an icon file.



v 0.7

Open Issues

- Where is the metadata associated with the Root directory located? Currently, root metadata is located in the root version table. All other files and directory metadata can be found in their parent directory.

eStream FSRFD Low Level Design

Sanjay M Pujare
[redacted]

Functionality

The File System and Registry Filter Driver (FSRFD) is a part of the Builder module that monitors file system and registry updates initiated by the Builder process. This driver just intercepts such requests and records them and returns the recorded data to the Install Monitor (*INSTALLMON* described in another LLD) program when requested by the latter. All the intelligence, such as any decision-making logic, resides in the INSTALLMON.

For registry updates such as add or modify, the FSRFD needs to record the value added or modified. For registry deletes only the value name needs to be recorded. For file updates, there is no need to record the file contents added or modified, since the Builder would be interested only in the final contents of a file.

Note:

1. **This does not cover those rare cases where an existing file is updated by an application install, and the eStream client would need to make the same updates. This kind of functionality is difficult to implement and will not be considered for 1.0.**
2. **The FSRFD will be used to monitor only one install at a time to simplify the design of the FSRFD. That means you cannot invoke multiple instances of the Builder at a time to monitor multiple installations. All Builder invocations on a machine have to be strictly sequential.**
3. **This design is based on the driver model for WinNT and Win2K. The Win98 driver is not covered here (yet).**

Data type definitions

The following struct is used to communicate information related to activating the FSRFD. Specifically, the process-id of the INSTALLMON and the 2 drives whose accesses need to be monitored are passed.

```
struct MonitorActivate_t {
   ULONG processId; // PID of INSTALLMON
   UCHAR sysDrive;  // System drive letter
   UCHAR destDrive; // Dest drive letter
};
```

The following struct is used to return monitored data back to the INSTALLMON. Note that this is a variable size struct where the last field keyName is an array of one wide-char, but in reality is an array of length whose value is the sum of 3 length fields in the struct (nameLength, valueNameLength, and dataLength).

```
struct IMON_ENTRY {
   UCHAR regOrFile; // 'R' for registry, 'F' for files
                       // and 'E' for end of data
   UCHAR updateType; // 'A' for add, 'D' for delete,
                       // 'U' for update
   UCHAR valueType;  // for registry only: value type
// REG_SZ, REG_DWORD, REG_BINARY,
// REG_DWORD_LITTLE_ENDIAN, REG_DWORD_BIG_ENDIAN,
// REG_EXPAND_SZ, REG_LINK, REG_MULTI_SZ, REG_NONE,
// REG_QWORD, REG_QWORD_LITTLE_ENDIAN,
// REG_RESOURCE_LIST
   ULONG nameLength; // length of name (file or
                       // registry key) in wchars
   ULONG valueNameLength; // length of value name (if
                             // it exists) in wchars
   ULONG dataLength;      // length of data in bytes
   WCHAR keyName[1];     // keyName followed by
                       // valueName followed by
                       // data: note none of these are
                       // null terminated & are wide
                       // chars
};
```

Note about the updateType field: 'A' is used for file creation and 'U' is used for any updates to the file. So if a 'U' is seen without an 'A' for a file that means the file was modified but not created in this session.

The following struct is used as the device extension in the FSRFD devices. Note that this extension is used for all device objects: the device that is created in the DriverEntry function to represent an "INSTALLMON" device for the INSTALLMON to access our driver as well as the devices that are created to create a filter layer above existing drives.

```
    enum DEVICE_TYPE {
              INSTALLMONINTERFACE,
              STANDARD
    };
```

```
struct DeviceExtension_t {
          PDEVICE_OBJECT deviceObject; // device
                         // for lower layer
          DEVICE_TYPE type; // see design
          KMUTEX pDeviceMutex;
          ProcessIdList
          // pointer to list of process-ids we are
          // interested in monitoring
          EntryList
          // This is the list that stores all the
          // info captured by the FSRFD until each
          // entry is queried by INSTALLMON
};
```

There is an array for FastIo that stores all the FastIo function pointers which is required in a file system filter driver such as this. Note that we need to provide entry points for all (or most?) of the FastIo routines in the dispatch table, since we need to pass the request down to the lower layer driver even if we are not interested in intercepting the request. For only some of the requests (e.g. all the FASTIO_*WRITE* requests), we would be recording the file access and creating an entry to be returned to INSTALLMON. The Component Design section describes in more detail the FastIO routines that are implemented by this driver.

Interface definitions

INSTALLMON interfaces

Since the FSRFD is a driver, it cannot provide directly callable APIs. Instead the INSTALLMON communicates with the FSRFD using the DeviceIoControl Win32 API. It uses Ioctl codes MON_ACTIVATE, MON_DEACTIVATE and MON_GET_ENTRY. The DeviceIoControl API looks as follows:

```
BOOL DeviceIoControl(
  HANDLE hDevice,              // handle to device
  DWORD dwIoControlCode,       // operation control code
  LPVOID lpInBuffer,           // input data buffer
  DWORD nInBufferSize,         // size of input data buffer
  LPVOID lpOutBuffer,          // output data buffer
  DWORD nOutBufferSize,        // size of output data buffer
  LPDWORD lpBytesReturned,     // byte count
  LPOVERLAPPED lpOverlapped   // overlapped information
);
```

The semantics of each of the MON_* codes is defined below. Note that this is described from the caller's (i.e. INSTALLMON) point of view.

Ioctl 1 – MON_ACTIVATE

Input:

hDevice:
  is the handle to the FSRFD device created.

dwIoControlCode:
  The value MON_ACTIVATE

lpInBuffer:
  Address of MonitorActivate_t struct. This contains the process id of INSTALLMON.

nInBufferSize:
  Sizeof(MonitorActivate_t)

Output:

lpOutBuffer:
  Should be a ptr to a ULONG (at least).

nOutBufferSize:
  Size of above buffer

lpBytesReturned:
  Should be a ptr to a DWORD where byte count of data returned in lpOutBuffer is returned.

Comments:

MON_ACTIVATE is sent to the FSRFD when the INSTALLMON wants to start monitoring an installation. MON_ACTIVATE can be sent only when the FSRFD is not already active - either after the driver is loaded the first time or after the last MON_DEACTIVATE request.

Errors:

- STATUS_ALREADY_ACTIVE: FSRFD was already activated. The processId of the old activation is returned in the ULONG pointed by lpOutBuffer.
- STATUS_INVALID_ARG: One of the arguments passed is not valid (either invalid MonitorActivate_t ptr or processId).

- STATUS_INVALID_DRIVE: Either the sysDrive or the destDrive (or both) is invalid.

Ioctl 2 – MON_DEACTIVATE

Input:

hDevice:
is the handle to the FSRFD device created.

dwIoControlCode:
The value MON_DEACTIVATE

lpInBuffer:
NULL, or pointer to a ULONG where the ULONG has a non-zero value indicating a "forced" deactivation. A "forced" deactivation is done when the FSRFD still has entries that are not going to be retrieved.

nInBufferSize:
0 or size of ULONG (depending on the above).

Output:

No need for OUT arguments.

Comments:

MON_DEACTIVATE is sent to the FSRFD when the INSTALLMON wants to stop monitoring an installation. MON_DEACTIVATE can be sent only when the FSRFD is already active i.e. after the last MON_ACTIVATE request.

Errors:

- STATUS_NOT_ACTIVE: FSRFD was already deactivated. No special action is needed to be taken for this error condition. The caller can simply send a new MON_ACTIVATE.
- STATUS_PENDING_DATA: The FSRFD has some entries that were not read using

MON_GET_ENTRY. This error is only returned in case of non-forced deactivation.

Ioctl 3 – MON_GET_ENTRY

Input:

hDevice:
is the handle to the FSRFD device created.

dwIoControlCode:
The value MON_GET_ENTRY

lpInBuffer:
NULL.

nInBufferSize:
0

Output:

lpOutBuffer:
Should be a ptr to a IMON_ENTRY struct.

nOutBufferSize:
Size of above buffer

lpBytesReturned:
Should be a ptr to a DWORD where byte count of data returned in lpOutBuffer is returned.

Comments:

MON_GET_ENTRY is sent to get the next "entry" from the FSRFD. An "entry" is a record of a registry or file update intercepted by the FSRFD. The details are returned in the IMON_ENTRY struct passed in the lpOutBuffer argument. Note that the FSRFD uses an event object (created using the IoCreateNotification-Event API) to signal the INSTALLMON that an "entry" is available to be read. INSTALLMON waits on this event object before retrieving the entry using MON_GET_ENTRY.

Errors:

- STATUS_NOT_ACTIVE: FSRFD was not activated.
- STATUS_INVALID_ARG: One of the pointer arguments passed is not valid.
- STATUS_INSUFF_BUFFER: The buffer size indicated by nOutBufferSize is insufficient for the current "entry" data.

Ioctl4 – MON_GET_ERROR

We need this to indicate occurrence of an error whenever this occurs in any of the dispatch functions. We can either implement this control code or just return an error code for any MON_GET_ENTRY call that reflects that an error occurred.

Event Object interface

As mentioned above, an event object (lets call it IMON event object) will be used to signal the INSTALLMON that a new entry is available. This event object will be created in the FSRFD in the processing of MON_ACTIVATE ioctl, as:

```
ext->pEvent = IoCreateNotificationEvent(
  L"\\BaseNamedObjects\\INSTALLMONEVENT",
        ext->eventHandle);
```

Whenever the FSRFD has a new entry, it signals using the above event object as follows:

```
KeSetEvent(ext->pEvent, 0, FALSE);
```

Whenever INSTALLMON gets the next entry using MON_GET_ENTRY ioctl, the FSRFD resets the event (if the list of entries is going to be empty after this entry) as follows:

```
KeClearEvent(ext->pEvent);
```

Whenever a MON_DEACTIVATE is processed, the FSRFD will close the event as:

```
ZwClose(ext->eventHandle);
```

Kernel or Low Level Driver Interfaces

Every driver needs a `DriverEntry` routine that is called when the driver is first loaded. This routine for FSRFD is described in the Component Design section.

The FSRFD inserts "hook routines" that intercept relevant registry and file system calls. The Component Design section describes which hook routines are inserted and what they do.

Component Design

Global variables

<u>pImonDevice</u>

This global variable points to the device object created in the DriverEntry function for the "\\Device\\installmon" device.

<u>DriverEntry</u>

```
NTSTATUS DriverEntry(IN DriverObject, IN RegistryPath)
{
            IoCreateDevice for "\\Device\\installmon";
            pImonDevice = pDeviceObject returned above;
            ext = pImonDevice->DeviceExtension;
            ext->type = INSTALLMONINTERFACE;
            IoCreateSymbolicLink with
                "\\DosDevices\\installmon";
            for all IRP_MJ_* values upto
                  IRP_MJ_MAXIMUM_FUNCTION {
               DriverObject->MajorFunction[IRP_MJ_*] =
                     ImonDispatch
            }
            Setup the unload driver function
            DriverObject->FastIoDispatch = address of
               our fast io dispatch table;
               Note that we are interested only in
               FAST_IO_*WRITE* routines for getting our
               entries, however we need to implement all
               of them to call lower layered drivers. All
               our FastIo funcs are called ImonFastIo*;
            Create the necessary mutexes;
            Use PsSetCreateProcessNotifyRoutine to set a
            process create callback routine
            ImonProcessCallback;
}
```

<u>ImonProcessCallback</u>

```
{
            // similar to ImonProcessCallback
            // This function is called every time a
            // process on the system is created or
            // deleted. We need to figure out (by
            // checking the parent id in our list)
            // if we need to add or remove this process
            // from our list
            lock the mutex for ProcessIdList
            if not activated just return;
            if (this is process create) {
               Look up the parent process id in the
               ProcessIdList
               If present, add this process id to the
                  ProcessIdList
            }
            else {
               if this process id is present then
                  remove the process id
            }
            release the mutex
      }
```

ImonDispatch

```
      {
            // similar to FilemonDispatch
            // Instead of registering a different
            // function for each IRP_MJ_* this function
            // is called for all of them and this one
            // dispatches the right on based on the
            // control code
            This gets called for all IRP_MJ_*;
            if the device extension type tells us
               INSTALLMONINTERFACE
                  call ImonDeviceFunc
            else
                  call ImonHookFunc
      }
```

ImonDeviceFunc

```
      {
            // similar to FilemonDeviceRoutine
            // This function is called whenever an Ioctl
            // comes from the Installmon process that is
            // meant to be a command for this FSRFD.
            This is a request from the INSTALLMON using
            the INSTALLMON device. We would mainly be
            processing IRP_MJ_DEVICE_CONTROL, although
            ImonFastIoDeviceControl should have been
            called. So if we come here just call
```

```
                ImonFastIoDeviceControl
                Call IoCompleteRequest?
    }
```

ImonHookFunc

```
    {
                // similar to FilemonHookRoutine
                // This is the hook function that is called
                // for all the I/O requests that is made by
                // the I/O manager that need to go through
                // this driver (i.e. for those requests
                // that we are filtering).
                We are interested in recording:
                  IRP_MJ_CREATE where the Irp->
                    Parameters.Create.Options indicates
                    a new file create (as opposed to an
                    existing file open)
                  IRP_MJ_WRITE
                Note that we have to get the current
                process-id using PsGetCurrentProcessId and
                look it up in the ProcessIdList (note: you
                have to exclude the first process-id since
                that belongs to the Installmon and not the
                setup program) and only if that search is
                successful, record the entry
                Note that we need to get the filename from
                  the FileObject using code similar to
                  FilemonGetFullpath
                Also we should be recording the entry when
                the request is successfully completed. So do
                this in the completion routine in a manner
                similar to filemon.
                To record:
                  create a relevant IMON_ENTRY record;
                  Call ImonAddEntry with this record;
                Pass all Irps to lower layer driver using
                  IoCallDriver and getting the lower layer
                  device object from this device's
                  ext->deviceObject
    }
```

ImonFastIoDeviceControl

```
    {
                // similar to both
                // FilemonFastIoDeviceControl and
                // ImonDispatchIoctl
                // This function gets called for all the
                // IOCTLs. If it is from Installmon, we need
                // to process the MON_* commands or else
                // just pass on the command to the lower
                // layer driver.
                Get the current device's extension
                if type indicates INSTALLMONINTERFACE {
                   switch (IoControlCode) {
                      case MON_ACTIVATE:
                         call ImonActivate;
                         break;
                      case MON_DEACTIVATE:
                         Call ImonDeactivate;
                         break;
                      case MON_GETENTRY:
                         Call ImonGetEntry;
                         break;
                   }
                }
                else {
                   pass it down using deviceObject and the
                   fastio hook for Ioctl
                }
    }
```

ImonAddEntry

```
    {
                // similar to ImonEnqueueRegEntry and
                // createRegEntry etc.
                // This function adds a IMON_ENTRY node
                // to our list: this list is eventually
                // returned to InstallMon
                create an entry rec from nonPagedPool
                Grab a mutex to modify EntryList
                Add the entry rec to EntryList
                release the mutex
                KeSetEvent for IMON event object
    }
```

ImonActivate

```
    {
                // similar to ImonActivate and various
                // Filemon funcs called by
                // FilemonFastIoDeviceControl
                // This is called by our own func that
                // handles IOCTLs when a MON_ACTIVATE is
                // sent by InstallMon
                Look at the ProcessIdList to ensure that we
```

```
              are not already active (1st process id).
              If we are, return with an error with that
              process id
              Grab a mutex
              Add a node to ProcessIdList with the
              processId;
              Release the mutex;
              Create the IMON event object;
              HookDrive(sysDrive);
              HookDrive(destDrive);
              HookRegistry();
    }
```

ImonDeactivate

```
    {
              // Same as above except for MON_DEACTIVATE
              If we are already deactivated
                 return with an error;
              If EntryList is not empty and this is not
                 a forced deactivation then
                 return with error;
              Clear and Delete the IMON event object;
              Free the ProcessIdList;
              Free the EntryList;
              UnhookRegistry();
    }
```

ImonGetEntry

```
    {
              // When the InstallMon sends a MON_GET_ENTRY
              // this function is called.
              Grab the mutex to access EntryList
              get next entry and remove from the
              list;
              if no entry then {
                 if (deactivated and first process in
                      the processIdList is dead) {
                    Make a new entry of "end of data"
                       type;
                 }
                 else {
                    there is some problem (should not
                       happen)
                 }
              }
              copy the entry into the OUT buffer
              release the mutex;
    }
```

HookDrive, UnhookDrive

```
    {
              // very similar to Filemon's HookDrive
              // Hence not described here
              // Similarly UnhookDrive
              // When a MON_ACTIVATE comes, we need to
              // make sure that all the requests to the
              // system drive or dest drive are filtered
              // through us. e.g. If "C:" is the system
              // drive, HookDrive will register this
              // as the filter driver for all IO for "C:"
              // Similary for UnhookDrive.
    }
```

HookRegistry, UnhookRegistry

```
    {
              //similar to Installmon's (Un)HookRegistry
              // We need to make sure that all Registry
              // functions are replaced, with our funcs
              // so that these funcs get called whenever
              // a process is trying to access the
              // registry. Our hook funcs will in turn
              // call the real funcs after queueing an
              // entry
    }
```

ImonFastIo* routines

```
    {
              // These are very similar to FilemonFastIo
              // routines except that we need to intercept
              // only FAST_IO_WRITE,
              // FAST_IO_MDL_WRITE_COMPLETE,
              // FAST_IO_WRITE_COMPRESSED,
              // FAST_IO_WRITE_COMPLETE_COMPRESSED
              Call the lower layer driver;
              if the request was successfully completed {
                 Record the details into our entry rec
                 and enqueue it similar to how ImonHookFunc
                 does it;
              }
    }
```

Interesting issues to deal with:

- Make sure that we use non-paged memory as required. e.g. all the nodes of the processIdList and entryList will be from non-paged pool. According to Bob, we do not need to always allocate non-paged memory – for Registry related nodes (i.e. when it is 'R' type IMON_ENTRY node) we can allocate paged memory. However this needs to be checked – since there will be pointers between these nodes, we need to make sure that this will work properly.
- We have to make sure that a 2-phase install works with this: some setups ask you to reboot the machine and after the reboot the setup continues. For the FSRFD we need to make sure that after the reboot the FSRFD is already loaded before the 2nd phase of the install starts. In the FSRFD we may need to make sure that when the "Runonce" registry key is updated (the installer is trying to do a 2-phase install and setting the 2nd phase exe as the value of "Runonce") we capure that info and accordingly co-ordinate with the Installmon to do the right thing. To ensure that this driver is started at the right time on start-up (since the Builder machine is going to be an in-house dedicated machine, it is okay), we need to add to the System registry (under `HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services`) appropriate values for `Start`, `Group` and `Tag` (and possibly others) value names. Actually this is going to be implemented in InstallMon as a user initiated event in which case the user informs the Builder UI (which in turn informs the InstallMon) that a reboot is imminent. In that case the InstallMon can try to find all possible ways in which the setup.exe is achieving this:
  - The "RunOnce" key or one of its other incarnations (e.g. RunOnceService, RunOnceEx etc) has been modified. We need to figure out exactly which one of the "*RunOnce*" can be modified.
  - The setup.exe has actually added itself (or another exe) to the startup folder. If that is the case, we can do the same trick here: replace that entry with an entry pointing to the BuilderUI with the original setup.exe value as an argument to the BuilderUI.
- An issue that hasn't been resolved is any user interaction (and user input) that has taken place during installation that is being monitored. For example, an installation may ask for a port number that it may store in a registry key. The solution suggested is as follows: This has to be a manual process. The Builder user should record all the manual interaction and manual data input that has taken place. He should recreate the same interaction in the custom DLL that the appInstallBlock provides for that app. This custom DLL at eStream app subscribe time can do the same thing that was observed during original application install. Alternatively we can request the ISV's co-operation in doing this. (May be this bullet should be transferred to a different doc such as the InstallMon LLD.).
- There is another unresolved issue about handling h/w or s/w dependent things that the installer does: we will have to handle this case by case basis and any knowledge we gain as a result of this, we should consolidate in the Builder components. e.g. we may notice that some installations may depend on IE4 or IE5 being there. Of course, one of the pre-requisites of the Builder is that it will be run on a pristine machines, so that we capture a maximal installation when it is taking place.

Testing design

Unit testing plans

This will be unit tested using the INSTALLMON program (or its early prototype). The INSTALLMON program sends all the required Ioctls like MON_ACTIVATE, MON_DEACTIVATE and MON_GETENTRY. The INSTALLMON output will be used to check the correctness of the FSRFD. This means the INSTALLMON itself should be assumed to be correct.

In addition to the above, we actually need to write one or more test programs that exercise the FSRFD. These test programs will be run as if they were App Installers i.e. as child processes of the INSTALLMON. The test programs should be written to exercise:

- All file systems (FAT, FAT32, NTFS, HPFS, compressed drives and others), since the FastIo routines need to be tested.
- For each of the file systems:
    - File create (with and without the old file being there)
    - File update (existing file appended as well as modified in the middle)
    - File touch (e.g. get the version of a DLL file) – this is not yet covered by this design
- Registry updates:
    - create a key
    - delete a subkey
    - delete a named value from a key
    - RegReplaceKey (we need to investigate if this is broken down into smaller reg calls).
    - RegRestoreKey (same comment as above applies)
    - RegSetKeySecurity (we failed to address this in the design)
    - RegSetValueEx
    - RegLoadKey and RegUnLoadKey

Stress testing plans

The FSRFD will be stress-tested using the above testing strategy by varying the rate at which files/registry are updated and under a variety of conditions (memory/disk, other processes).

Coverage testing plans

The unit testing above also covers Coverage testing.

Cross-component testing plans

This will be tested with the Installmon program which is enough for cross-component testing.

Upgrading/Supportability/Deployment design

Deployment: This will be used in-house, so no deployment considerations.

Open Issues

These issues are not necessarily FSRFD related, but are listed here just to remind ourselves.

- Do we need to worry about environment variables? It looks like most installations (and their apps) won't be looking at environment variables.
- What about the .lnk files (shortcuts). It looks like the client guys will have to change all the .lnk files based on the actual client's settings. This issue has been addressed either in the InstallMon or Packager. Pls see those documents. (The `IShellLink` interface has been suggested).
- Also what about when device drivers are installed? There is no impact on the builder but the client will need to reboot and hopefully the existing appInstallBlock and the custom initialization dll should be able to take care of it.

eStream Installmon Low Level Design

Sanjay M Pujare
<Date>

Functionality

The Installmon is a part of the Builder module that talks to the FSRFD to monitor file system and registry updates initiated by the Builder process. The FSRFD driver just intercepts such requests and records them and returns the recorded data to Installmon when requested by the latter. All the intelligence, such as any decision-making logic, resides in the Installmon.

Note:

1. **This does not cover those rare cases where an existing file is updated by an application install, and the eStream client would need to make the same updates. This kind of functionality is difficult to implement and will not be considered for 1.0.**

2. **Somewhere in the docs (may be the user docs?) it should be mentioned that the Builder's (or rather Installmon's) job could be made easier by the user by following these guidelines:**

   - **Make sure that values in all the registry keys are as distinct as possible. We may need to create a special Win2K or WinNT installation where each key (or ValueName within a key) will be created with a distinct or unique value as far as possible. E.g. If the default Windows installation creates 2 keys FOO and BAR and stores the same value "C:\Windows\System32" in both of them, we wouldn't know which one of those keys is used when a file is copied to "C:\Windows\System32". To solve this problem, all the effort should be made to ensure that FOO and BAR have distinct values.**

   - **When the Builder operator is installing an app under the Installmon, she should also make sure that the install script is given a distinct or unique value for each of the user inputs that may be used to set a registry key or an environment variable. This is especially necessary when the inputs seem to be totally unrelated. For example, vendor name and installation directory name. If both are entered as "Microsoft" then that could cause confusion to the Installmon.**

3. There are 2 ways in which registry changes and file system changes can be captured:

   - ❑ using a kernel mode driver such as the FSRFD, Or
   - ❑ using a diff'ing mechanism for both the registry as well as the file system.

- **In this design, we use both the approaches, and record any differences due to these two approaches. We give the user a chance to manually edit the registry/file changes.**

Data type definitions

Interface definitions

All of the interfaces used to communicate with the FSRFD are described in the FSRFD LLD document. This LLD document will cover only interfaces with the other modules.

Interfaces with the Builder UI

The public methods of the InstallMon class described below are interfaces to the Builder UI. Specifically these are:

```
void InstallMon::startCapture(PUNICODE_STRING setup_exe,
   PUNICODE_STRING dest, bool upg, FileTable_t *fTbl);
```

This function is called when the user chooses to start monitoring an install. This function is called after the user has entered all the necessary data such as the setup.exe path etc.

```
void InstallMon::stopCapture()
```

This is called in rare cases when the user wants to terminate a running install in abnormal cases.

```
bool InstallMon::checkSetupStatus()
```

This is used by the UI to check the status of the install: whether the file list is ready to be processed and the registry list is ready to be processed etc.

```
void InstallMon::getRegistryList()
```

This function is used to finally get the list of registry changes that occurred after the install has taken place. This function allows the user to edit the list to manually add/delete/modify entries to be able to override.

```
void InstallMon::getFilesList()
```

This function is used to finally get the list of file system changes that occurred after the install has taken place. This function allows the user to edit the list to manually add/delete/modify entries to be able to override.

```
void InstallMon::machineToBeRebooted()
```

When the app install program is asking the user to reboot the machine (in the middle of the install) to continue installation, the user has to inform the Builder UI that a reboot is imminent. The InstallMon does the necessary bookkeeping to make sure that monitoring continues after the reboot.

```
void InstallMon::startCaptureAfterReboot(setup_exe, dest,
upg)
```

When the installation continues after the reboot, the Builder UI is automatically invoked, and it calls this function to inform the InstallMon to continue monitoring.

Access DB interface

We will be using an Access DB (or alternately the MSI databases as suggested by Bhaven) to store intermediate results of the Installmon process. There will be 2 tables used: Registry and Files.

Registry table

```
Fullpath :string;          // this is the full path of the key
ValueName :string;         // this is the value name: for
            // (default) use NULL
UpdateType: char;          // Add/update or delete, also
            // 'R' for removed
// combined (Fullpath, ValueName) should be unique i.e.
// primary key
```

Files table

```
Fullpath : string;
UpdateType : char;         // add, update
Kind: char;                // 'C'opied, 'S'poofed, 'E'stream
FileId: Number;
// Combined (Fullpath, UpdateType, FileId) should be unique
```

RegistryDiff table

```
Fullpath :string;
ValueName:string;
ValueType: char;
Value: something that can store all REG_* types
status: char;   // 'O'riginal, 'C'(add/change)., 'D'
// combined (Fullpath, ValueName) should be unique
```

FilesDiff table

```
Fullpath: string;
Type: char;     // file 'F' or dir 'D' etc
Status: char;     // 'O'riginal, ...'C', 'A', 'D' etc
// (Fullpath
```

Component Design

```
struct FileTable_t {
  PUNICODE_STRING name;
  Char type; // spoofed, copied or eFS
  Int  fileId;
};

class InstallMon {
```

```
PUNICODE_STRING setup, destDrive;
bool interruptThread;
EventObject signalMonitor, signalSetup;
bool rebootReq = false;
bool afterReboot = false;
bool completed;
int exitCode;
bool upgrade;
someType envVars;
// TODO: combine the above 2 into an enum
int upgradeFileIdBegin = -1;
int currFileId;
static threadSetup(InstallMon iMon) {
  Remove all the environment variables except
  the ones in iMon->envVars;
  Start the Setup program;
  And wait for it to finish;
  iMon->exitCode = exit code from the process;
  when finished signal the signalSetup event;
}
static threadMonitor(InstallMon *iMon) {
     // this is the func that runs as a separate
     // thread, until we are interrupted or we
     // notice that the setup process has exited.
  get the current
  process id, system drive (using
  GetSystemWindowsDirectory), destDrive and send
  the MON_ACTIVATE message>
  Start threadSetup thread with setup_exe;
  processEnded = false;
  while (!interruptThread) {
        if (rebootReq) {
          only poll for Installmonevent;
          if (signal not set) {
            break from the while loop;
          }
        }
        else {
          WaitForMultipleObjects -> signalMonitor,
          Installmonevent and signalSetup;
        }
        switch (signal) {
        case Installmon event:
          get the MonitorEntry_t record;
          switch (regOrFile) {
          case 'R':
            switch (updateType) {
```

```
            case 'A':
            case 'U':
               add or update (KeyName, ValueName,
                   'N') to
                  registry table;
               break;
            case 'D':
               add (KeyName, ValueName, 'D') to
                  registry table;
         }
         break;
      case 'F':
         switch (updateType) {
            case 'A':
               // file creation
               add (fullpath, 'A', '?',
                  iMon->currFileId++) to files
                  table;
               break;
            case 'U':
               add (fullpath, 'U') to files
                  table;
               break;
         }
         break;
      case 'E':
         no more data to add;
         if (!processEnded) {
            // something wrong!!!!
         }
         break out of the while loop
         break;
      }
      break;
   case setupevent:
      processEnded = true;
      Send MON_DEACTIVATE to the FSRFD;
      break;
   case signalMonitor event:
      if (rebootReq) {
         kill the threadSetup thread
         clear the signalMonitor event;
      }
      else {
         // TBD?
      }
      break;
```

```
            } // switch
        } // while
        if (processEnded) {
            // normal setup process completion
            // registry and files tables should have
            // all the captured data from the FSRFD
            iMon->diffCapture();
            iMon->completed = true;
        }
    } // threadMonitor
    void commonCapture(setup_exe, dest) {
        setup = setup_exe;
        destDrive = dest;
        Start the threadFunc thread, and pass 'this' to
        it;
    }
    void diffCapture() {
      // Bhaven suggested that we could instead do an export
      // from regedit and do a text diff for registry diffs.
      // similarly: It seems the regular Unix diff tool also
      // compares directories. I don't know if it is
      // recursive. If it is, we might be able to use it.
      // Further investigation is recommended for doing
      // files diff.
      Query the OTI\BUILDERSTART key and get the value
      in builderStartTime and delete the
      OTI\BUILDERSTART key;
      // process the registry
      // step 1: query RegistryDiff table
      for each row in RegistryDiff table {
        query the corresponding key+valueName in the
        Windows registry
        if (exists) {
          if (no change to value) {
            update the row status to 'U' for unchanged
          }
          else {
            update the row status to 'C' (changed)
          }
        }
        else {
          update the row status to 'D' (deleted)
        }
      }
      // step 2: enumerate the whole Windows registry
      for each enumerated registry key+value {
        query the RegistryDiff table;
        if (a row exists) {
            if (status is 'U') {
                delete the row
            }
            else {
                status should be 'C' and values should be
                different bet table and actual registry
                or else display fatal error(?)
            }
        }
        else {
            // no previous value
            add a row to RegistryDiff with (fullpath,
                valueName, ValueType, Value, 'A') to mark
            it as an added registry key
        }
    }
    // now repopulate the FilesDiff table
    // step 1: query FilesDiff table
    for each row in FilesDiff table {
        query the corresponding Fullpath in the actual
        file system;
        if (exists) {
            if (no change (i.e. timestamp before
                builderStartTime)) {
                update the row status to 'U' for unchanged
            }
            else {
                update the row status to 'C' (changed)
            }
        }
        else {
            update the row status to 'D' (deleted)
        }
    }
    // step 2: traverse the whole file system
    for each file/dir in {systemDrive, destDrive} {
        query the FilesDiff table;
        if (a row exists) {
            if (status is 'U') {
                delete the row
            }
            else {
                status should be 'C' and the file/dir
                timestamp should be after builderStartTime
                or else display fatal error(?)
            }
            }
            else {
               // no previous value
               add a row to FilesDiff with (fullpath,
                  'F' or 'D', 'A') to mark it as an added
               file/dir;
            }
         }
      }
      void recursiveFileGetter(curDir) {
         add a row to FilesDiff as (curDir, 'D', 'O');
         for (each element of curDir) {
            if (element is a dir) {
               recursiveFileGetter(element);
            }
            else { // has to be a file(?)
               add a row to FilesDiff as (element, 'F',
                     'O');
            }
         }
      }
   public:
      InstallMon(?);
      void setEnvVars() {
         allow the user to set environment variables
         in an interactive way;
         Get the values in envVars;
      }
      void startCapture(PUNICODE_STRING setup_exe,
            PUNICODE_STRING dest, bool upg, FileTable_t *fTbl) {
         clear the Access DB registry, files,
            registryDiff and FilesDiff tables;
         if (upg) {
            populate the files table with data from the
               fTbl array;
            upgradeFileIdBegin = last file id + 1;
            currFileId = upgradeFileIdBegin;
         }
         else {
            currFileId = 0 or 1 (depends on where to start);
         }
         Query the whole registry and
         for (each key and valueName pair) {
            add (key, valueName, valueType, value, 'O') to
               registryDiff;
         }
         Get the current time and store it in some format
      in a registry key OTI\BUILDERSTART;
      for (curDrive in {systemDrive, dest}) do
      {
         Root = root of curDrive (e.g. "C:\\");
         recursiveFileGetter(Root);
      }
      interruptThread = false;
      afterReboot = false;
      commonCapture(setup_exe, dest, upg);
   }
   void stopCapture() {
         // abnormal capture termination
         // TBD
         // also there might be normal cases where this
         // func is called when the setup process exit
         // is not detected for some reason (or doesn't
         // happen)! May be we don't have to worry about
         // these things.
   }
   bool checkSetupStatus() {
      if (!completed) {
         display error and return false;
      }
      if (exitCode is not okay) {
         display error and return false;
      }
   }
   void getRegistryList() {
      checkSetupStatus() and conditionally return;
      Using appropriate SQL, show (fullpath, ValueName)
        tuples that exist in Registry but not in
        RegistryDiff;
      Tell the user that these were captured by FSRFD
      (installmon) driver but not by the diff process;
      If user chooses to remove any tuples from the
      shown list, mark these with status as 'R' in the
      Registry table;
      Using appropriate SQL, show (fullpath, ValueName)
        tuples that exist in RegistryDiff but not in
        Registry;
      Tell the user that these were captured by diff
      but not by the FSRFD (installmon) driver;
      If user chooses to add any tuples from the
      shown list, add these tuples to the Registry
      table with status copied from RegistryDiff;
      Remove all the rows from the Registry table
      where status is 'R';
```

```
    Now we have all the correct entries in Registry;
    Read each row of Registry and into an array to be
    passed to the caller (most probably the caller of this
    func would have passed an array in which we should
    pass these rows);
  }
  void getFilesList() {
    checkSetupStatus() and conditionally return;
    Using appropriate SQL, show (fullpath)
      tuples that exist in Files but not in
       FilesDiff;
    Tell the user that these were captured by FSRFD
    (installmon) driver but not by the diff process;
    If user chooses to remove any tuples from the
    shown list, mark these with status as 'R' in the
    Files table;
    Using appropriate SQL, show (fullpath)
      tuples that exist in FilesDiff but not in
       Files;
    Tell the user that these were captured by diff
    but not by the FSRFD (installmon) driver;
    If user chooses to add any tuples from the
    shown list, add these tuples to the Files
    table with status copied from FilesDiff;
    Remove all the rows from the Files table
    where status is 'R';
    Now we have all the correct entries in Files;

    Using appropriate SQL, select files (and not
    dirs) where the file has only 'U' and not 'A':
    this means the setup modified an existing file
    and we cannot handle this; so if this happens
    show a fatal error;

    Read each row of Files and classify it into 'C',
    'S' or 'E' based on the following logic:
    If the file belongs to the app install directory
    (or app drive?) it is an 'E';
    If the file is smaller than a threshold then
    it is 'C' or else it is 'S';

    if (upg) {
      we need to create new file-ids for directories
      that contain new files or directories
      int prevStart = upgradeFileIdBegin;
      int prevEnd = currFileId;
      while ((prevEnd - 1) > prevStart) {
```

```
      using appropriate SQL, select rows from the files
      table where fileId >= prevStart and kind is not
         'C';
      for each such row {
         fullpath = fullpath in the row; (e.g. "C:\A\B")
         dir = parent directory of fullpath (e.g.
               "C:\A")
         select in files a row where fullpath = dir and
         fileId >= upgradeFileIdBegin;
         if (doesn't exist) {
            add in Files a row with (dir, 'X', '?',
               currFileId++);
            // Here 'X' stands for "new file id for a
            // directory created because one of the
            // children has a new file id
         }
      }
      prevStart = prevEnd;
      prevEnd = currFileId;
   }
}
```

Also for each file, change the absolute path to an envVar relative or registry-value relative one: this is a non-trivial task. The way to do this is proposed below. However we need to refine this algorithm based on actual Builder runs on real apps:

For a basic Win2K (or WinNT as the case may be) system create a list of registry keys (and env vars) whose values are normally used as destination paths for copying various kinds of files. To this list also add this app's dest dir as one of the values. Also add all the registry keys/values added as part of this app's install. Create an access table that looks like:

```
Key: String;     // registry or env or other name
Type: char;     // 'R'egistry, 'E'nv, 'D'est dir etc.
Source: char;  // 'S'ystem, 'A'pp, 'U'nknown?
Value: string;
```

Now all of the 'E' files should be relative to dest dir (i.e. 'D' type above). Also any sub-directory strings should either be hard-coded or based on some registry key values where the registry key is of source 'A' (app) above.

```
    All the 'C' and 'S' files should be relative to one of
    the registry keys of source 'S' above and preferably
    type 'R' (since environment variables are not used un-
    der Windows?).

    If there is a sacred file set (i.e. files that are
    so "sacred" for eStream apps, that they were installed
    on the client when the eStream client was installed),
    we need to remove those files from this list;

    Set these values in the table and read the whole
    table in an array (most probably the caller of
    this func would have passed an array in which we
    should pass these rows);
}
void machineToBeRebooted() {
     //The setup is going to reboot the machine.
     //Note that there is a race condition possible here.
     // Suppose the setup.exe has displayed a dialog
     // asking the user to confirm a reboot. At this time
     // the setup.exe has still not written to "Runonce".
     // Only after the user has confirmed (by pressing
     // Okay) will the program write to "runonce" and
     // auto reboot. In that case, this function will be
     // called at the wrong time. We may need to modify
     // the FSRFD to detect changes to "Runonce".
     rebootReq = true;
     signal the signalMonitor event;
     wait for the threadMonitor thread to end;
     Query either our Access database or the actual
     registry to see if Runonce key is set/updated.
     if (not) {
        we cannot handle this reboot situation;
        give fatal error;
        // may be we can look at the
        // Start\Programs\Starup folder and do the
        // same thing we did with Runonce key
     }
     OldRunonce = Old value of Runonce key;
     Set the new value of Runonce key as our Builder
     name (or whichever EXE has the installmon code)
     followed by OldRunonce as the argument to that
     EXE and the destDrive as the next argument and
     upgrade as the next arg;
     Prompt the user to go ahead and say Okay to
     Setup's dialog warning that it is going to
     reboot;
    }
    void startCaptureAfterReboot(setup_exe, dest, upg) {
        afterReboot = true;
        commonCapture(setup_exe, dest, upg);
    }
};
```

Interesting issues to deal with:

Testing design

Unit testing plans

The unit testing of installmon will be done in conjunction with the FSRFD unit testing. This has been described in the FSRFD-LLD document. In addition, we will be using the sysdiff tool to validate the results of the Installmon.

Stress testing plans

All of the 14 or so applications (that OTI is planning to convert to eStream) installs will be tested under the installmon. Any issues found will be used to fix and improve the installmon functionality.

Coverage testing plans

The testing of the Builder/installmon on the 14 or so app installs should give us enough coverage. Considering that Builder/installmon will be used in-house for some time makes some of the testing issues less significant.

Cross-component testing plans

Will be tested as a component of the whole builder.

Upgrading/Supportability/Deployment design

Deployment: This will be used in-house, so no deployment considerations.

Open Issues

- On one of the newgroups someone mentioned a key HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\SharedDLLs] that we can look at for the shared DLLs that cannot be installed. Need to investigate.
- 2-phase install: some installs need a reboot after which they continue. The key to look at is Runonce (under the same path as SharedDLLs I think).

eStream 1.0 Client

Registry Spoofing Database

Omnishift Technologies, Inc.
Company Confidential

Functionality

This component is a device driver that is started and stopped by the eStreamClient.exe program.

The registry-spoofing database is a list of registry entries that the registry spoofer redirects whenever client programs or the windows kernel requests spoofed registry keys. The reason that we do this is to present different views of the registry depending on when different users are logged into the system. Upon completion of User Login a startup program called eStreamControl is begun to initialize the following client components (not necessarily in this order).

1. Registry Spoofing
2. File Spoofing
3. Z File System Set up
4. eStream Logon and user Verification.

Registry spoofing is begun by a call into an eStream Device Driver from the eStreamControl program that will start the kernel level registry spoofer. When the user logs out the eStreamControl will make a call to the eStream Device Driver to shut down the registry spoofing. If another user logs on to the same computer who is not enabled as an eStream user then that users view of the registry is identical to what it would be if eStream were not installed on the client computer.

Data type definitions

The registry entries to be spoofed are provided on a per-user set. Every user who logs on to a computer with eStream installed will have their own set of spoofed registry entries. These registry entries will be stored under HKEY_CURRENT_USER. This will enable simple back up of the keys when the user logs out. The HKEY_CURRENT_USER Keys are automatically backed up to HKEY_USERS when the user logs out and restored when the user Logs back in.

Keys that need to be spoofed are always in one of the following places

HKEY_CLASSES_ROOT (HKCR)
HKEY_LOCAL_MACHINE (HKLM)
HKEY_CURRENT_CONFIG (HKCC)

Spoofed Keys are stored in the registry database under HKEY_CURRENT_USER. This portion of the registry database is set up when a user logs on and is stored back into the HKEY_USERS when the user logs off.

The App install block uses a .reg file. The keys in the .reg file are added to the registry by the regedit program provides.

The structure of the keys will be as follows.

```
HKEY_CURRENT_USER
        Estream
                Registry Spoof
                        Add
                                HKCR
                                    ...
                                HKLM
                                    ...
                                HKCC
                                    ...
                        Remove
                                HKCR
                                    ...
                                HKLM
                                    ...
                                HKCC
                                    ...
```

The eStream Registry spoofer needs to be able to delete keys as well as overwrite keys and add keys. The Overwrite and create key functions are provided by the Add Sub Keys. The Remove key may be a very sparse tree, but some software units will probably need the facility.

Components

Spoof Device Driver

This is a device driver that has a DEVICE_IOCTL command to toggle registry spoofing on and off.

Spoof Regedit Utility Program

The utility program Regedit is not aware that registry spoofing is going on. It would be very helpful if an enhanced version of Regedit was developed that will be aware of registry spoofing and will identify keys that are added, deleted, or modified by the registry spoofing.

eStreamControl

This is an executable program that is started when the user logs on. The function of this program is to start up the eStream services that are required by the user to run eStream Applications. This program also shuts down eStream application services when the user logs out.

Interface definitions

Registry Spoofing uses the same interfaces that the registry uses. The only interface that is required is a device driver call to toggle registry spoofing on and off. The actual registry spoofer in the Windows Kernel will go to HKEY_CURRENT_USER\eStream to find its spoof database.

StartRegSpoofing()

Causes the registry spoofer to begin spoofing using the current database.

StopRegSpoofing()

Component design

The registry spoofing device driver will intercept all of the following kernel level registry calls.

RtlCheckRegistryKey

```
NTSTATUS
  RtlCheckRegistryKey(
  IN ULONG  RelativeTo,
  IN PWSTR  Path
  );
```

RtlCheckRegistryKey checks for the existence of a given named key in the registry.

RtlCreateRegistryKey

```
NTSTATUS
  RtlCreateRegistryKey(
  IN ULONG  RelativeTo,
  IN PWSTR  Path
  );
```

RtlCreateRegistryKey adds a key object in the registry along a given relative path.

RtlWriteRegistryValue

```
NTSTATUS
  RtlWriteRegistryValue(
  IN ULONG  RelativeTo,
  IN PCWSTR  Path,
  IN PCWSTR  ValueName,
  IN ULONG  ValueType,
  IN PVOID  ValueData,
  IN ULONG  ValueLength
  );
```

RtlWriteRegistryValue writes caller-supplied data into the registry along the specified relative path at the given value name.

RtlDeleteRegistryValue

```
NTSTATUS
  RtlDeleteRegistryValue(
  IN ULONG  RelativeTo,
  IN PCWSTR  Path,
  IN PCWSTR  ValueName
  );
```

RtlDeleteRegistryValue removes the specified entry name and the associated values from the registry along the given relative path.

```
NTSTATUS
  ZwCreateKey(
  OUT PHANDLE  KeyHandle,
  IN ACCESS_MASK  DesiredAccess,
  IN POBJECT_ATTRIBUTES  ObjectAttributes,
  IN ULONG  TitleIndex,
  IN PUNICODE_STRING  Class  OPTIONAL,
  IN ULONG  CreateOptions,
  OUT PULONG  Disposition  OPTIONAL
  );
```

Path

Specifies the registry path according to the *RelativeTo* value. If RTL_REGISTRY_HANDLE is set, *Path* is a handle to be used directly. If the path intercepts any spoofed registry path then the path will be redirected to HKEY_CURRENT_USER\eStream.

RtlQueryRegistryValues

```
NTSTATUS
  RtlQueryRegistryValues(
  IN ULONG  RelativeTo,
  IN PCWSTR  Path,
  IN PRTL_QUERY_REGISTRY_TABLE  QueryTable,
  IN PVOID  Context,
  IN PVOID  Environment  OPTIONAL
  );
```

RtlQueryRegistryValues allows the caller to query several values from the registry subtree with a single call.

All of these Kernel level Registry functions need to be intercepted. If the path variable intercepts any path inside HKEY_CURRENT_USER\estream then the call will be redirected to that key.

Private Helper Function

BOOL IsPathSpoofed([in] PCWSTR Path, [out] PCWSTR eStreamPath);

This will return TRUE if the path intercepts an eStream Key, FALSE otherwise. If the function returns TRUE then the corrected Spoof path will be placed in the eStreamPath argument.

Re-Registration of Objects

Many application programs such as Office Applications will register themselves every time they execute on the client machine. The Registry Spoofer will need to intercept these writes and re-direct them to the HKEY_CURRENT_USER\eStream set of registry keys.

Testing design

Generating Reg Key files and then using the normal regedit program to determine if the keys are being returned correctly will test this unit. Adding and removing registry keys using a client program that checks the spoofed path to the key can test this component quickly.

The registry-spoofing diver will need to be tested with the following Windows SDK Registry Functions.

Registry Functions

The following functions are used with the registry.

Function	**Description**
RegCloseKey	Releases a handle to the specified registry key.
RegConnectRegistry	Establishes a connection to a predefined registry handle on another computer.
RegCreateKeyEx	Creates the specified registry key.
RegDeleteKey	Deletes a subkey.
RegDeleteValue	Removes a named value from the specified registry key.
RegDisablePredefinedCache	Disables the predefined registry handle table of **HKEY_CURRENT_USER** for the specified process.
RegEnumKeyEx	Enumerates subkeys of the specified open registry key.
RegEnumValue	Enumerates the values for the specified open registry key.
RegFlushKey	Writes all the attributes of the specified open registry key into the registry.
RegGetKeySecurity	Retrieves a copy of the security descriptor protecting the specified open registry key.
RegLoadKey	Creates a subkey under **HKEY_USERS** or **HKEY_LOCAL_MACHINE** and stores registration information from a specified file into that subkey.
RegNotifyChangeKeyValue	Notifies the caller about changes to the attributes or contents of a specified registry key.
RegOpenCurrentUser	Retrieves a handle to the **HKEY_CURRENT_USER** key for the user the current thread is impersonating.

RegOpenKeyEx	Opens the specified registry key.
RegOpenUserClassesRoot	Retrieves a handle to the **HKEY_CLASSES_ROOT** key for the specified user.
RegOverridePredefKey	Maps a predefined registry key to a specified registry key.
RegQueryInfoKey	Retrieves information about the specified registry key.
RegQueryMultipleValues	Retrieves the type and data for a list of value names associated with an open registry key.
RegQueryValueEx	Retrieves the type and data for a specified value name associated with an open registry key.
RegReplaceKey	Replaces the file backing a registry key and all its subkeys with another file.
RegRestoreKey	Reads the registry information in a specified file and copies it over the specified key.
RegSaveKey	Saves the specified key and all of its subkeys and values to a new file.
RegSetKeySecurity	Sets the security of an open registry key.
RegSetValueEx	Sets the data and type of a specified value under a registry key.
RegUnLoadKey	Unloads the specified registry key and its subkeys from the registry.

The following are the initialization-file functions. They retrieve information from and copy information to a system- or application-defined initialization file. These functions are provided only for compatibility with 16-bit versions of Windows. New applications should use the registry.

Function	**Description**
GetPrivateProfileInt	Retrieves an integer associated with a key in the specified section of an initialization file.
GetPrivateProfileSection	Retrieves all the keys and values for the specified section of an initialization file.
GetPrivateProfileSectionNames	Retrieves the names of all sections in an initialization file.
GetPrivateProfileString	Retrieves a string from the specified section in an initialization file.
GetPrivateProfileStruct	Retrieves the data associated with a key in the specified section of an initialization file.
GetProfileInt	Retrieves an integer from a key in the specified

	section of the Win.ini file.
GetProfileSection	Retrieves all the keys and values for the specified section of the Win.ini file.
GetProfileString	Retrieves the string associated with a key in the specified section of the Win.ini file.
WritePrivateProfileSection	Replaces the keys and values for the specified section in an initialization file.
WritePrivateProfileString	Copies a string into the specified section of an initialization file.
WritePrivateProfileStruct	Copies data into a key in the specified section of an initialization file.
WriteProfileSection	Replaces the contents of the specified section in the Win.ini file with specified keys and values.
WriteProfileString	Copies a string into the specified section of the Win.ini file.

Obsolete Functions

These functions are provided only for compatibility with 16-bit versions of Windows.

RegCreateKey
RegEnumKey
RegOpenKey
RegQueryValue
RegSetValue

Unit testing plans

Unit testing can be performed with a simple client program that reads and writes keys in Spoofed Key folders.

Stress testing plans

Normal Windows operation reads and writes keys in high volume. One possible stress test would be to load and unload keys from the HKEY_CURRENT_USER/eStream/ database while simultaneously reading and writing the corresponding sub key folders.

Coverage testing plans

Test adding and removing keys from each of the spoofed key sets.

Add a key to each of the following points

HKEY_CURRENT_USER/eStream/Add/HKCR
HKEY_CURRENT_USER/eStream/Add/HKLM
HKEY_CURRENT_USER/eStream/Add/HKCC

Read the keys back from their corresponding keys in the following points.

HKEY_CLASSES_ROOT
HKEY_LOCAL_MACHINE
HKEY_CURRENT_CONFIG

Key deletion will also need to be checked. To accomplish this a set of keys is added to each of the following key folders

HKEY_CLASSES_ROOT
HKEY_LOCAL_MACHINE
HKEY_CURRENT_CONFIG

Keys are added to the corresponding Key Folders

HKEY_CURRENT_USER/eStream/Remove/HKCR
HKEY_CURRENT_USER/eStream/ Remove /HKLM
HKEY_CURRENT_USER/eStream/ Remove /HKCC

A client application reads the keys using the windows SDK functions

> **RegLoadKey**
> **RegOpenKey**

Cross-component testing plans

The install manager needs to install key sets using a .reg file. Using the regedit program can check this to see if the keys have been installed correctly.

eStream 1.0 Client UI Modules

Omnishift Technologies, Inc.
Company Confidential

Requirements

Here is a list of the eStream Client UI requirements

1. Administration of the Application Subscription and installation process
2. Cache and File system Management
3. License Management
4. User Account Log in
5. Application Syncing

Users running the eStream system need a simple system to subscribe and unsubscribe applications. The user interface that provides for this will come through the web browser. A browser plug in ActiveX control can provide a simple way for ASP web pages to interact with the user client system.

The File system driver will be loaded automatically when the user logs on. The Cache manager provides efficient data streaming to the file system driver. The Cache Manager is a worker thread inside the Client UI program.

License management provides tokens that will allow a user to access specific application files from an eStream server. The License Manager is a set of threads running inside the eStream Client program that acquires, renews, and releases access tokens as a subscribed application runs.

In order to access application executable and data files from an eStream Server a user must log onto an ASP server. The client UI program will provide this account management.

Users can access eStream from different computers at different times. Changes in Application subscription status synchronized between these different computers when the user logs on.

Functionality

The eStream Client UI module supports reporting eStream-specific error & informational messages to the client user and solicits replies when appropriate. It allows the eStream client user to view and change the list of applications currently installed on the client system and the list of ASP accounts currently known to the client system.

The eStream client will have a tray icon that will allow the user to access administration and control functions. Tray icons are small icons on the right side of the desktop toolbar. These tray icons give the user the indication that the eStream client is running and allow the user to access administrative functions that infrequently used.

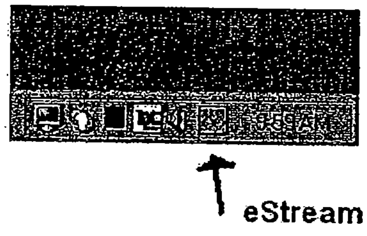


Figure 1 Tray Icon

Right clicking on the tray icon produces a pop up menu that would allow the eStream user to access the following functions.

- Set the cache size and review cache performance and utilization
- Log the user into and out of ASP Accounts
- Set the proxy server IP address
- Set the ASP server
- Access the ASP web page and view accounting and utilization information.

Normal eStream The App Install manager will update Start Menus and desktop shortcuts in a manner identical to a native installation of the same application. The user does not interact with the eStream client when running subscribed applications.

Components

eStreamClient.exe

This program is located in the startup folder and will be run when the user logs onto a system with at least one eStream subscribed applications.

- Log on the ASP
- Start the file Spoofer
- Start the EFS file system
- Start the License Manager and the Cache Manager
- Get run tokens from the LRM
- Contact the ASP(s) and update changes in subscribed application sets

When the initialization is complete the eStreamClient.exe program will wait for the user to log off the system. When the user logs off the eStream device drivers (registry and file spoofers) are halted, LRM tokens and released, and the eStreamClient.exe Program exits.

Browser Plug In

An OLE Automation client is the simplest way for a web page to access internal configuration parameters on a Microsoft Windows Client. An OLE automation Client will allow a Browser script, VB Program, or VC program to communicate directly with the Client executable program.

For a browser Script the eStream Client executable could be controlled using an object tag.

Here is an example of HTML code that uses and Object Tag to connect to an OLE Automation server.

```
<OBJECT ID="ESTREAMCLIENT"

 CLASSID="CLSID:D8D77E03-712A-11CF-8C70-00006E127B7"

CODEBASE=http://www.aspcorp.com/eStreamClient.exe

DATA="InstallSetASPOffice1.ODS"?

</OBJECT>
```

Interfaces provided by the eStream Browser Plug In

AppInstall (String AppInstallBlockPATH)

The App install block contains a header section, variable section, and other blocks. The App Install block is large so a file link is sent to the AppInstall interface where the AppInstall block has been installed. This will probably be in some temporary folder.

The AppInstall function will need to build an Uninstall block that will list the components of the subscribed application. Uninstalling applications cleanly and reliably is as important as important as installing them. The behavior of the App install manager should mimic the Add/Remove panel from the Windows Control panel.

AppUninstall(AppId)

Uninstall the application. The header file of the app install block that comes with the application when it was originally subscribed provides the AppId.

XML data islands working together with Active Server Pages would provide a simple and reliable system for administrating Applications.

Application Database

A database containing application subscription information needs to be maintained on the client. This database can be displayed back to the user using XML tags. This database needs to contain enough information to provide the following functions the complete uninstall of subscribed installation, accounting information, and access to ASP accounts.

Design of the Client UI Program

The eStreamClient program provides all of the run time functionality that the client requires. The modules absorbed by this unit include

- Client Log On
- App Install Manager
- License Subscription Manager
- Client Network Manager
- Cache Manager

The App Install manager works in conjunction with the ASP web page to allow users to subscribe and un-subscribe applications. The coordination between the browser plug-in and the App Install Manager will probably require some kind of a COM interface into the eStreamClient program.

The License Subscription Manager manages access tokens. Access Tokens are generated by the eStream Server for a limited period of time and must be renewed when they expire. The eStream Client UI has a set of threads that deal with this process.

The Client Network Manager is the portal through which all communication between cache manager, license subscription manager, and the eStream server flows.

The Cache Manager together with the file system driver is the core component of the eStream product. These two components serve application executables over the network to mimic native installation of application programs.

eStream 1.0 eStream Client Network Component

Omnishift Technologies, Inc.
Company Confidential

Functionality

The client network component communicates with the following servers for the types of requests listed.

Account server

Validate a user for this ASP and get subscription information

DRM server

1. Validate a license for a subscribed app

App server

1. Get an app install block for a subscribed app
2. Open a file/directory for a subscribed app
3. Various file requests on a previously opened file/directory

Data type definitions

Definitions provided here are only for data types that are shared among components; this will take coordination with these other components. Definitions should be as C-language structures, regardless of how they'll be implemented.

Interface definitions

Interfaces

Only a few file operation interfaces are listed

ValidateUser()

Input:

- account server to query
- user data to send, including perhaps
    - ASP identifier

    - username
    - password
    - client certificate

Output:

- subscription info for this user, including perhaps
    - currently subscribed apps
    - serial number for each app
    - DRM server to use for validation for each app

ValidateLicense()

Input:

- DRM server to query
- subscription serial number to validate
- client certificate

Output:

- access token for this licensed subscription
- app server to use for data requests

GetInstallBlock()

Input:

- app server to query
- subscription serial number
- client certificate

Output:

- app install block for this subscribed app

AppOpenFile()

Input:

- app server to query
- access token (possibly NULL)
- file name

Output:

- handle to use for further file requests

AppReadFile()

Input:

- app server to query
- access token
- file handle
- buffer to fill
- offset
- length

Output:

- filled buffer
- number of bytes read

UploadAppProfileDataRequest()

Input:

- account server to upload to
- app profile data

Output:

- success/failure

Component design

Testing design

Unit testing plans

Stress testing plans

Coverage testing plans

Cross-component testing plans

eStream Registry Spoofer current status

Introduction

Discussions about the need for active registry spoofing using a kernel level device driver and a complex database are presented here. There is a strong desire by the client design group to simply the design of the eStream software.

Reasons for Registry Spoofing

1. Leaving the registry pristine
2. Allowing multiple users to have different views of the same computer

One of the design goals of eStream is to provide a system of supplying applications to a user while having a minimal impact on that users system. The normal process of installing application software on a computer will make a large number of changes to the registry database on the client computer. One possible advantage of registry spoofing is keeping the client computers registry unmodified as applications are subscribed and un-subscribed.

Another advantage of registry spoofing is to allow different users of the same computer to have access to different versions of software packages. This could be an advantage in situations where individuals share computers and eStream applications subscriptions are granted to individual users not to all the users of a particular computer. Two different users could use differing versions of the same application. A single user who does not subscribe to eStream could use one version and another user through an eStream application subscription could use a more current version of the same application.

Reasons for not doing Registry Spoofing

1. Increased complexity of the eStream client software
2. Difficulty of testing
3. Adding complexity to the app install process.

The difficulty of creating a kernel level registry spoofer is well understood by the members of the engineering team that have worked on Windows device drivers. The Registry is a component of the operating system that is critical to the proper functioning of both the Windows operating system kernel and most application software. It is accessed frequently by both the kernel and application software and any malfunction in its operation would have devastating consequences to the reliability of the eStream product. There are over 33 Windows SDK functions that access the registry directly and probably hundreds more that access it indirectly. Testing all of these functions would be a very difficult undertaking and would be critical to the success of the eStream product.

In addition to the complexity added by a kernel level registry spoofer the addition of a set of spoofed registry entries for each subscribed application would add additional overhead, for both testing and installation, for each application that is available through the eStream subscription process.

Other Spoofing Options

1. Normal Installation of eStream Subscribed Apps
2. Logon time manipulation of the registry database

Possible alternatives to spoofing the registry database include keeping the registry entries of subscribed applications in a separate database that would be added to the registry when the eStream client is started, when the user logs on, or at system boot time. These registry entries would be removed at logout or shutdown time.

The popular consensus is to no perform registry spoofing at all on the eStream Client. The major advantage of this approach is to simplify the client application subscription process. The normal application install program that comes with each application could be used to subscribe an application if some way could be found to force the installer to place the application executables and portable data files on the EFS file system.

Conclusions

We have decided to abandon registry spoofing.

eStream 1.0 Installation Manager

Omnishift Technologies, Inc.
Company Confidential

Functionality

The Installation Manager consists of the following sub-components

Installshield

Installshield is the industry standard for building installation sets for Microsoft Windows. Installshield will take a set of executables and data files and create a media installation. The Installshield environment provides a scripting language that will allow a high degree of customization of target installation. The essentials issues for any installation are.

1. How much of the application does the user wish to install?
2. Is the users system capable of running the application?
3. Where does the user wish to install the application?
4. Does the user have enough space to install the applications?

Installshield has a wizard that will set up a project. When the install shield program is compiled a media must be specified. The most common media types are floppy, CD Rom, and Web media builds. For eStream we may have to ask the clients to reboot the machine since we are installing kernel mode components that might need a reboot to take effect.

Install From the Web

This program is another product that is sold by Installshield that will take a complete installation set and create a single executable .exe that can be easily downloaded from a web site.

Uninstaller

Installshield will provide an uninstaller when it builds the install program.

Registry Settings

There are three ways that Installshield can patch the system registry.

1. Run regsvr32.exe on self-registering .dll files. When the uninstaller is run it will use regsvr32.exe /u to un-register the .dll file.
2. Patch the registry statically.
3. Patch the registry based on Installation Options from the install shield script program.

Artwork

The Installshield program for eStream will require a splash screen and possibly one or two other artwork components.

Data type definitions

The data that the Installation manager uses

1. Device Drivers
2. COM Libararies
3. COM Executables
4. Registry files

Interface definitions

Testing design

Testing of the eStream Installer must take place on computers when Visual Studio has not ever been installed.

Unit testing plans

Install the eStream Client using the binary file installer from InstallShield. Since the number of installation options will be kept to a minimum this test should not take very long.

Installation of eStream Binaries from the InstallShield Installer is the first step of testing all new revisions of eStream.

Stress testing plans

The installation should be tested on a wide variety of computers with special emphasis on testing the installation on computers that do not have Visual Studio installed on them.

Coverage testing plans

Need component list for generating this test plan

Cross-component testing plans

All client components are installed with this piece so we don't have too much to worry about here.

eStream 1.0 License Subscription Manager (LSM)

Omnishift Technologies, Inc.
Company Confidential

Requirements

The purpose of the License Subscription manager is to acquire and manage list of tokens that will allow the cache manager to access application executable and data files on remote eStream servers.

Functionality

This component is a set of worker threads that are part of the Client UI process.

The LSM manages the users subscriptions to the different ASP accounts. It is part of the client component downloaded on a client machine. The LSM starts running when the client component starts running and is always active when the client component is running. Users on a given machine establish a connection with the ASP account servers from which they have subscribed applications. Users can add and delete the applications that are subscribed from the ASP accounts. The LSM makes the appropriate calls to the account servers to perform those actions. It gets serial numbers for the applications that are being subscribed and deletes them for the applications being un-subscribed (which are all part of the ASP ID Block). When the users start running any of the subscribed eStream applications, the eStream file system first queries the LSM before servicing any requests. The LSM in turn gets the appropriate access tokens from DRM servers along with the identities of application servers that can be used to run the applications. It uses the client identification (serial number) obtained when the connection to the ASP was made. At the same time, the LSM can decide to cache the access tokens and the identities of the application servers and decide to serve them directly from its cache. The eStream Cache Manager informs the LSM when applications start and end. The LSM keeps track of when access tokens are expiring and can request for additional access tokens when applications are running and the current one is about to expire.